| L Number | Hits | Search Text | DB | Time stamp |
|---|---|---|---|---|
| 1 | 482 | heap and pile | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:40 |
| 2 | 0 | heap and pile and identif4 with root | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:20 |
| 3 | 0 | heap and pile and identif$4 with root | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:20 |
| 4 | 0 | heap and pile and identif$4 same root | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:21 |
| 5 | 2 | heap and pile and identif$4 same (root parent) | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:21 |
| 6 | 0 | heap and pile and identif$4 same (root parent) and unused | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:21 |
| 7 | 0 | heap and pile and identif$4 same (root parent) and un$used | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:22 |
| 8 | 23170 | add with operation | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:22 |
| 9 | 160 | add with operation and root with leaf | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:23 |
| 10 | 19 | add with operation and root with leaf and identif$4 with root | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:23 |
| 11 | 3 | add with operation and root with leaf and identif$4 with root and un$used | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:24 |
| 12 | 3 | add with operation and root with leaf and identif$4 with root and (un$used "not used") | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:24 |
| 13 | 17 | add with operation and root with leaf and identif$4 with root and (un$used "not used" available) | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:25 |
| 14 | 2 | add with operation and root with leaf and identif$4 with root and (un$used "not used" available) with leaf | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:25 |
| 15 | 2 | add with operation and root with leaf and identif$4 with root and (un$used "not used" available) with leaf and travers$4 | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:26 |
| 17 | 0 | add with operation and root with leaf and identif$4 with root and (un$used "not used" available) with leaf and travers$4 with root with leaf and heap | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:27 |
| 18 | 0 | add with operation and root with leaf and identif$4 with root and (un$used "not used" available) with leaf and travers$4 with root with leaf and pile | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:27 |
| 19 | 0 | add with operation and root with leaf and identif$4 with root and (un$used "not used" available) with leaf and travers$4 with root with leaf and public adj domain | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:28 |
| 20 | 0 | add with operation and root with leaf and identif$4 with root and (un$used "not used" available) with leaf and travers$4 with root with leaf and data adj structure | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:28 |
| 16 | 2 | add with operation and root with leaf and identif$4 with root and (un$used "not used" available) with leaf and travers$4 with root with leaf | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:32 |

| | | | | |
|---|---|---|---|---|
| 21 | 2 | add with operation and (root parent) with (leaf child$4) and identif$4 with (root parent) and (un$used "not used" available) with (leaf child$4) and travers$4 with root with leaf | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:33 |
| 22 | 16 | (root parent) with (leaf child$4) and identif$4 with (root parent) and (un$used "not used" available) with (leaf child$4) and travers$4 with root with leaf | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:33 |
| 24 | 0 | (root parent) with (leaf child$4) and identif$4 with (root parent) and (un$used "not used" available) with (leaf child$4) and travers$4 with root with leaf and add$4 and heap | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:35 |
| 23 | 16 | (root parent) with (leaf child$4) and identif$4 with (root parent) and (un$used "not used" available) with (leaf child$4) and travers$4 with root with leaf and add$4 | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 11:16 |
| 25 | 0 | heap and pile and identif$4 with (root parent) and (available un$used "not used") with (node leaf child$4) | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:42 |
| 26 | 38 | heap and pile and travers$4 | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:42 |
| 27 | 0 | heap and pile and travers$4 and (root parent) and (leaf child) with (available unused) | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:42 |
| 28 | 0 | 707/$.ccls. and heap and pile and travers$4 | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 10:43 |
| 29 | 10697 | remov$4 adj operation | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 11:17 |
| 30 | 203 | remov$4 adj operation and data adj structure | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 11:18 |
| 31 | 2 | remov$4 adj operation and data adj structure and remov$4 with root with value | USPAT; EPO; JPO; IBM_TDB | 2003/04/01 11:19 |

US006105018A

# United States Patent [19]

**Demers et al.**

[11] **Patent Number: 6,105,018**

[45] **Date of Patent: Aug. 15, 2000**

[54] **MINIMUM LEAF SPANNING TREE**

[75] Inventors: **Alan Demers**, Boulder Creek; **Alan Downing**, Fremont, both of Calif.

[73] Assignee: **Oracle Corporation**, Redwood Shores, Calif.

[21] Appl. No.: **09/049,285**

[22] Filed: **Mar. 26, 1998** 102(e).

[51] **Int. Cl.**[7] ..... **G06F 17/30**

[52] **U.S. Cl.** ..... **707/2**; 707/5; 707/4; 707/3; 707/1

[58] **Field of Search** ..... 707/2, 5, 4, 3, 707/1

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 5,701,467 | 12/1997 | Freeston | 707/100 |
| 5,781,906 | 7/1998 | Aggarwal et al. | 707/102 |
| 6,006,233 | 12/1999 | Schultz | 707/101 |

OTHER PUBLICATIONS

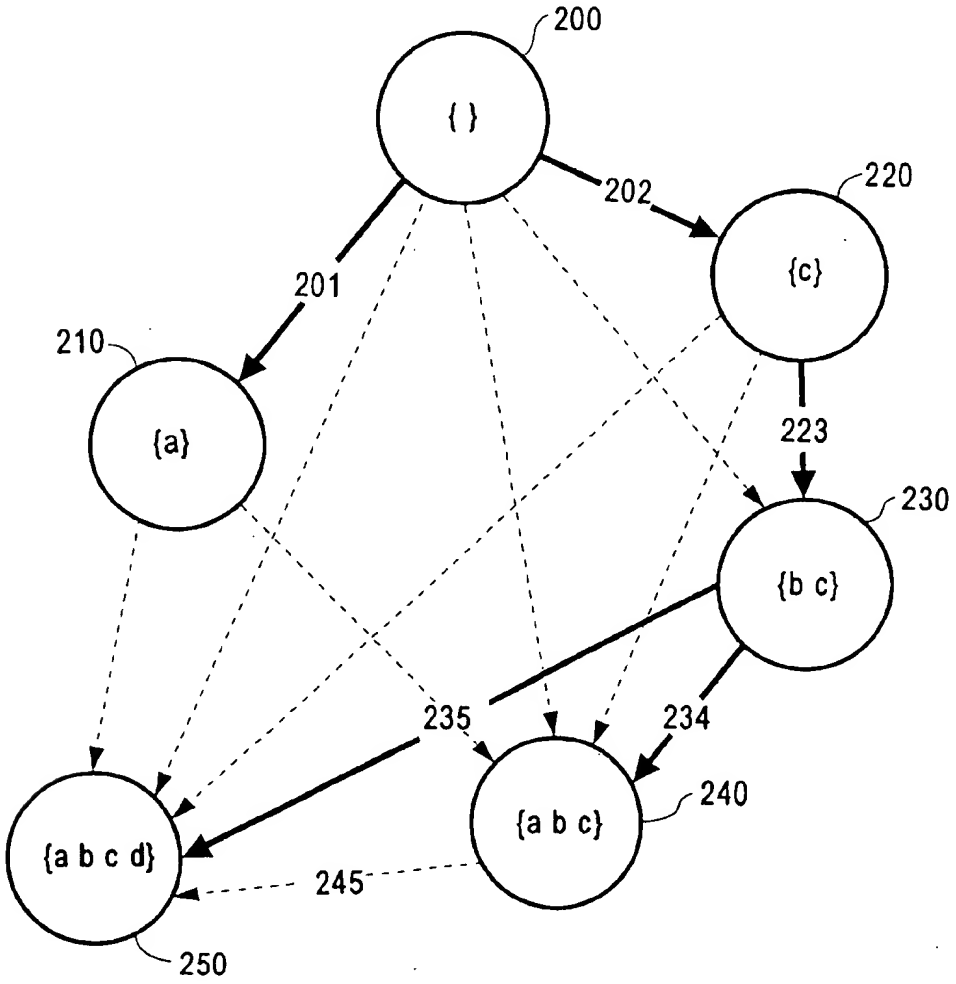
CRC Dictionary of of Computer Science, Engineering and Technology, Mar. 2000.

*Primary Examiner*—Thomas G. Black
*Assistant Examiner*—William Trinh
*Attorney, Agent, or Firm*—McDermott, Will & Emery

[57] **ABSTRACT**

An efficient set of indexes to cover a plurality of anticipated query types is determined by building a directed acyclic graph whose nodes correspond to anticipated query types. A minimum leaf spanning tree for the equivalent graph is determined by repeatedly finding an augmenting path for a current spanning tree and producing a reduced leaf spanning tree based on the current spanning tree and the augmenting path until an augmenting path can no longer be found. The leaves of the minimum leaf spanning tree indicate which indexes should be built.

**26 Claims, 12 Drawing Sheets**



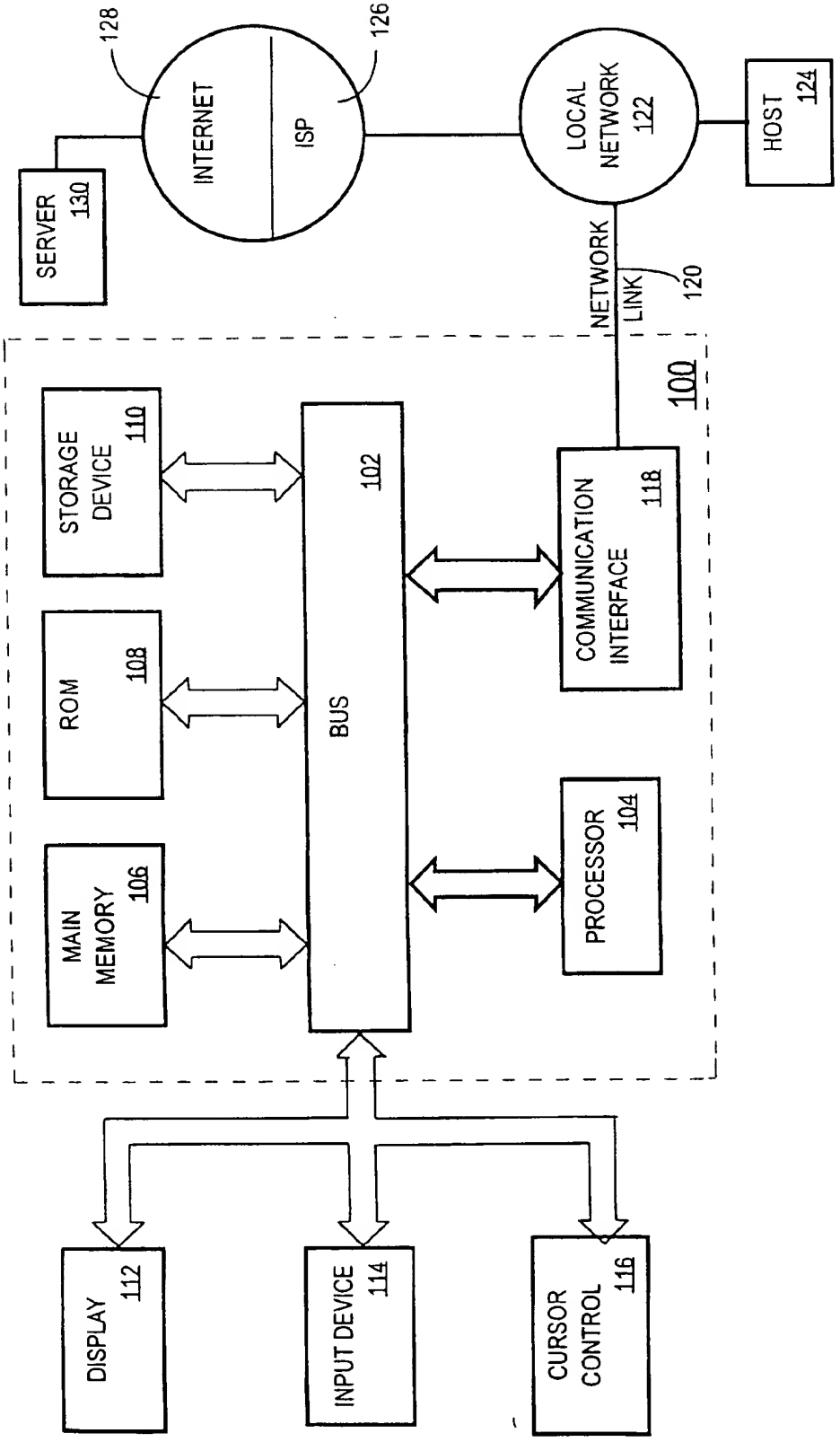

128
126
INTERNET
ISP
LOCAL NETWORK 122
HOST 124
SERVER 130
NETWORK LINK 120
100
STORAGE DEVICE 110
BUS 102
COMMUNICATION INTERFACE 118
ROM 108
PROCESSOR 104
MAIN MEMORY 106
DISPLAY 112
INPUT DEVICE 114
CURSOR CONTROL 116

FIG. 1

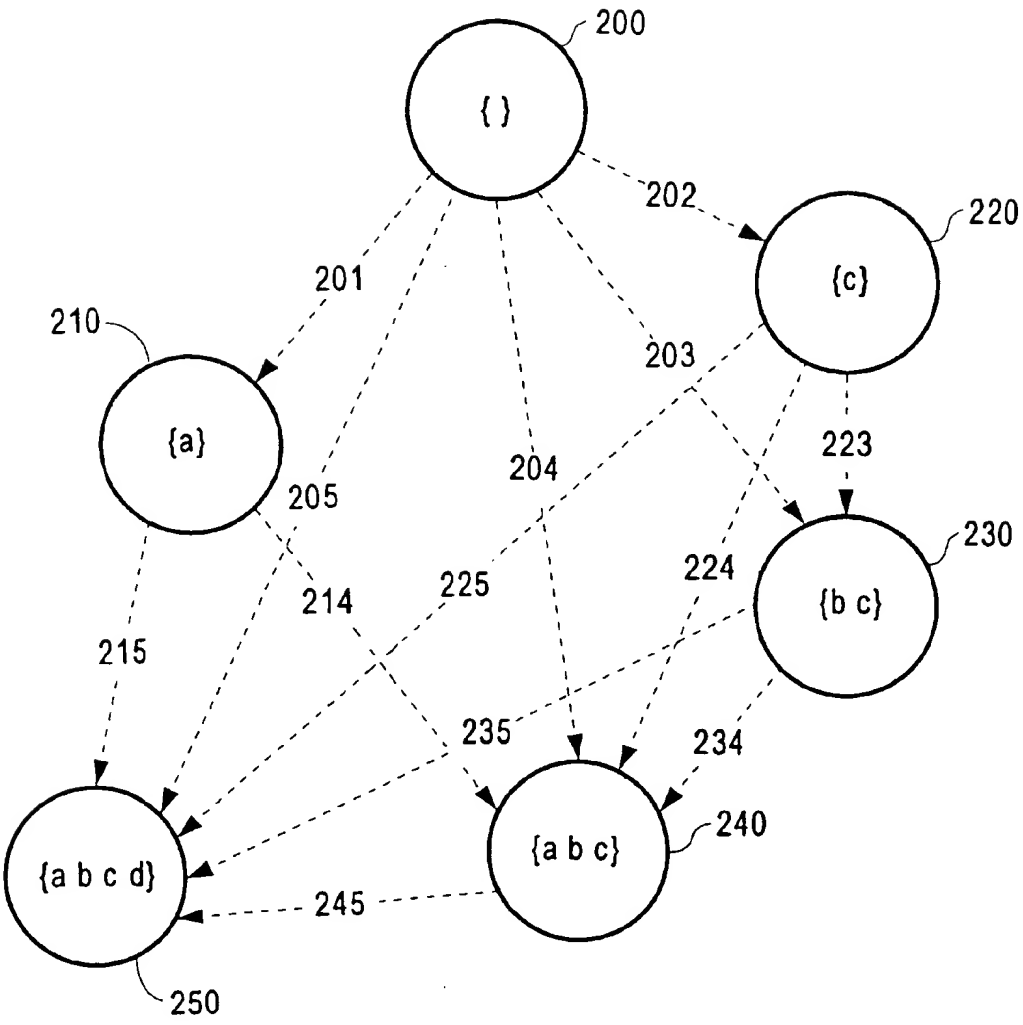

200
{ }
202
220
{c}
201
210
{a}
203
223
204
205
230
{b c}
224
214
225
215
235
234
{a b c d}
{a b c}
240
245
250

FIG. 2

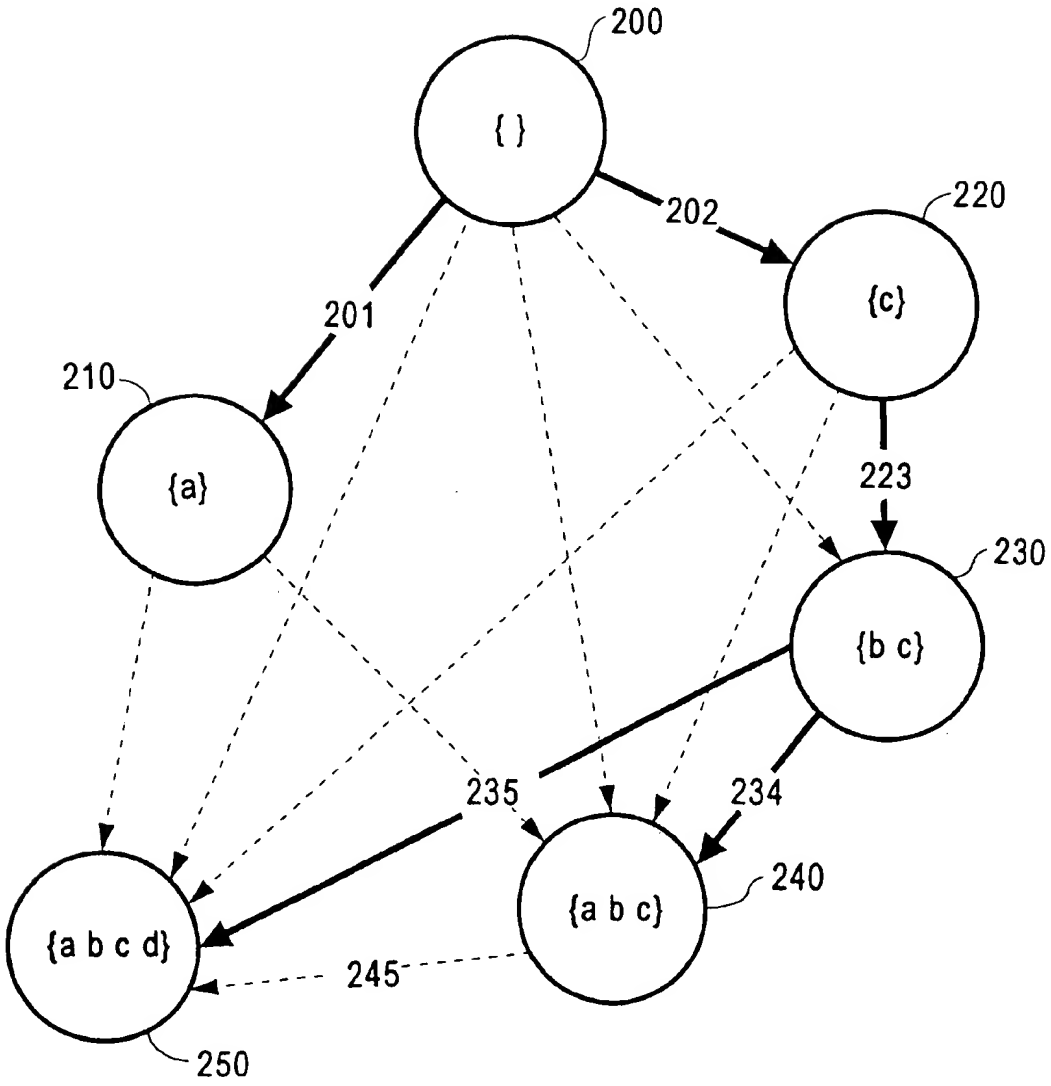



FIG. 3

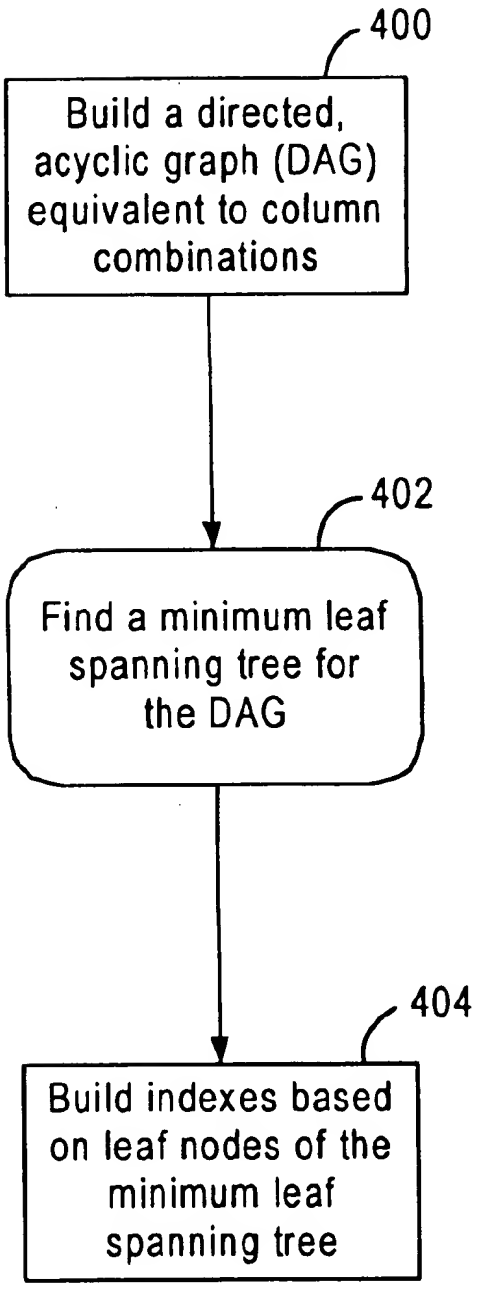

400
Build a directed, acyclic graph (DAG) equivalent to column combinations
402
Find a minimum leaf spanning tree for the DAG
404
Build indexes based on leaf nodes of the minimum leaf spanning tree

FIG. 4

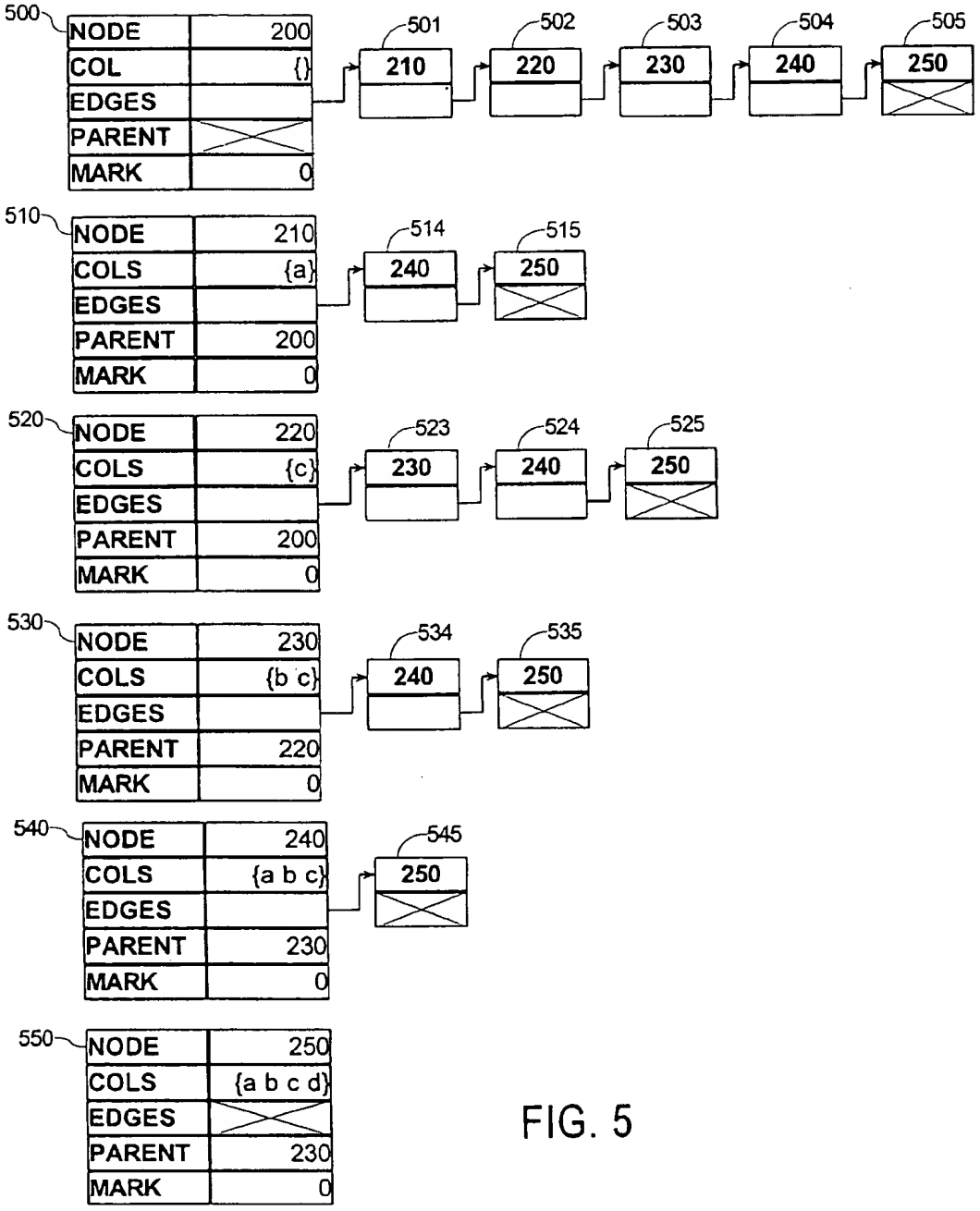

500
NODE 200
COL {}
EDGES
PARENT
MARK 0
501
210
502
220
503
230
504
240
505
250
510
NODE 210
COLS {a}
EDGES
PARENT 200
MARK 0
514
240
515
250
520
NODE 220
COLS {c}
EDGES
PARENT 200
MARK 0
523
230
524
240
525
250
530
NODE 230
COLS {b c}
EDGES
PARENT 220
MARK 0
534
240
535
250
540
NODE 240
COLS {a b c}
EDGES
PARENT 230
MARK 0
545
250
550
NODE 250
COLS {a b c d}
EDGES
PARENT 230
MARK 0

FIG. 5

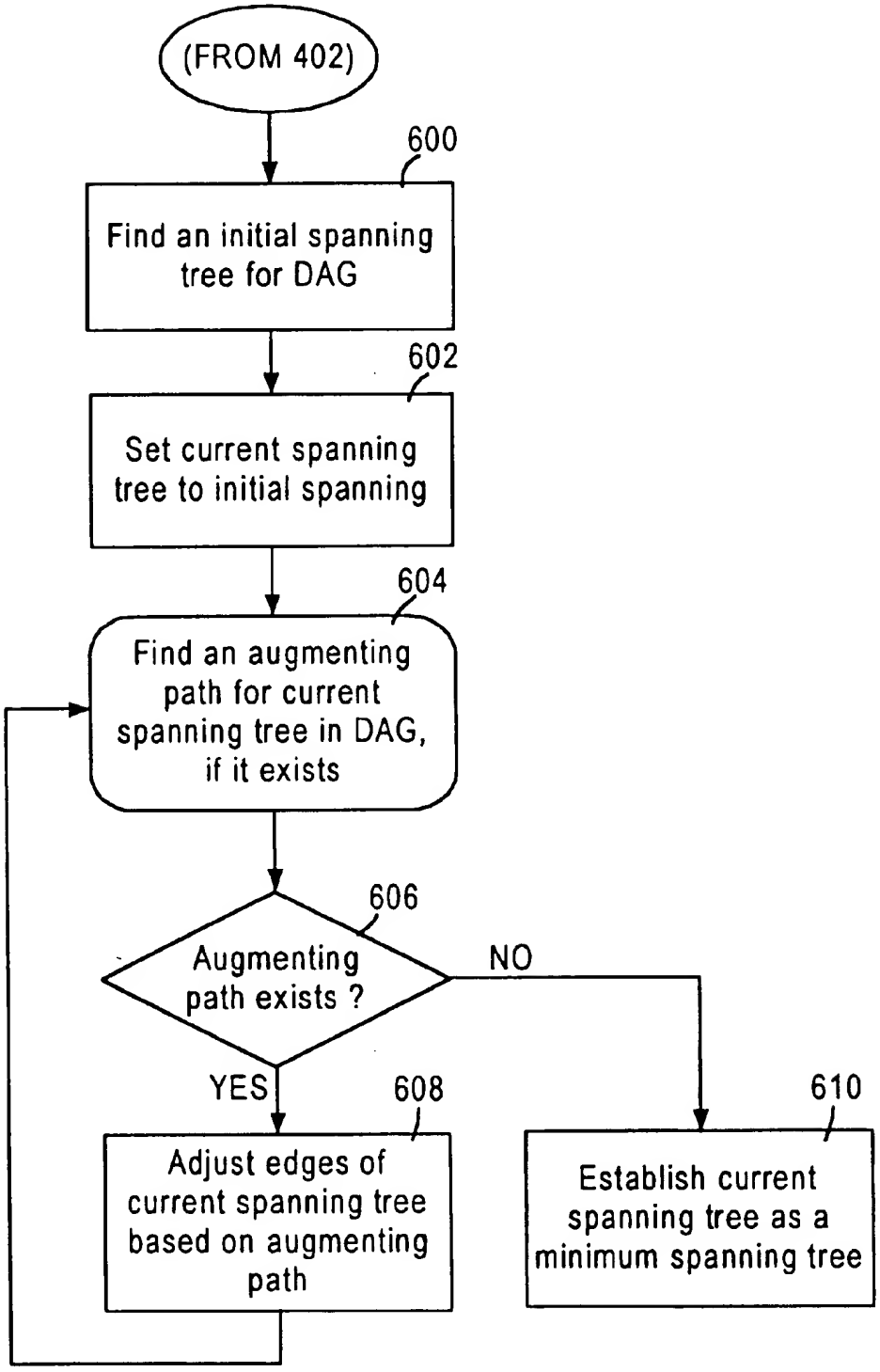

(FROM 402)
600
Find an initial spanning tree for DAG
602
Set current spanning tree to initial spanning
604
Find an augmenting path for current spanning tree in DAG, if it exists
606
Augmenting path exists ?
NO
YES
608
Adjust edges of current spanning tree based on augmenting path
610
Establish current spanning tree as a minimum spanning tree

FIG. 6

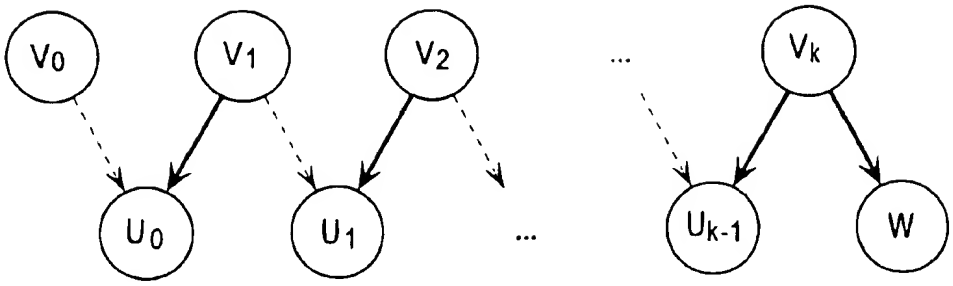



FIG. 7(a)

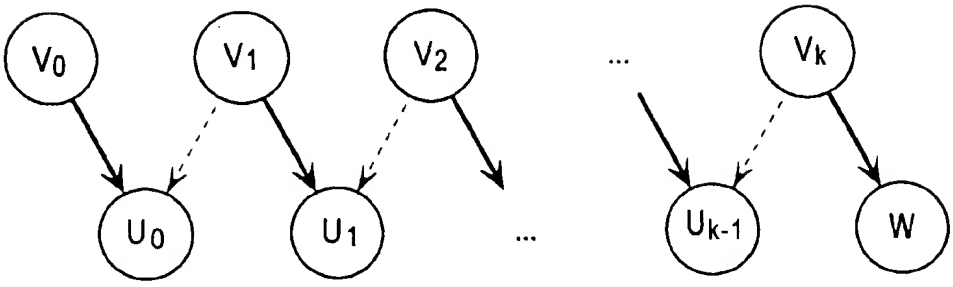



FIG. 7(b)

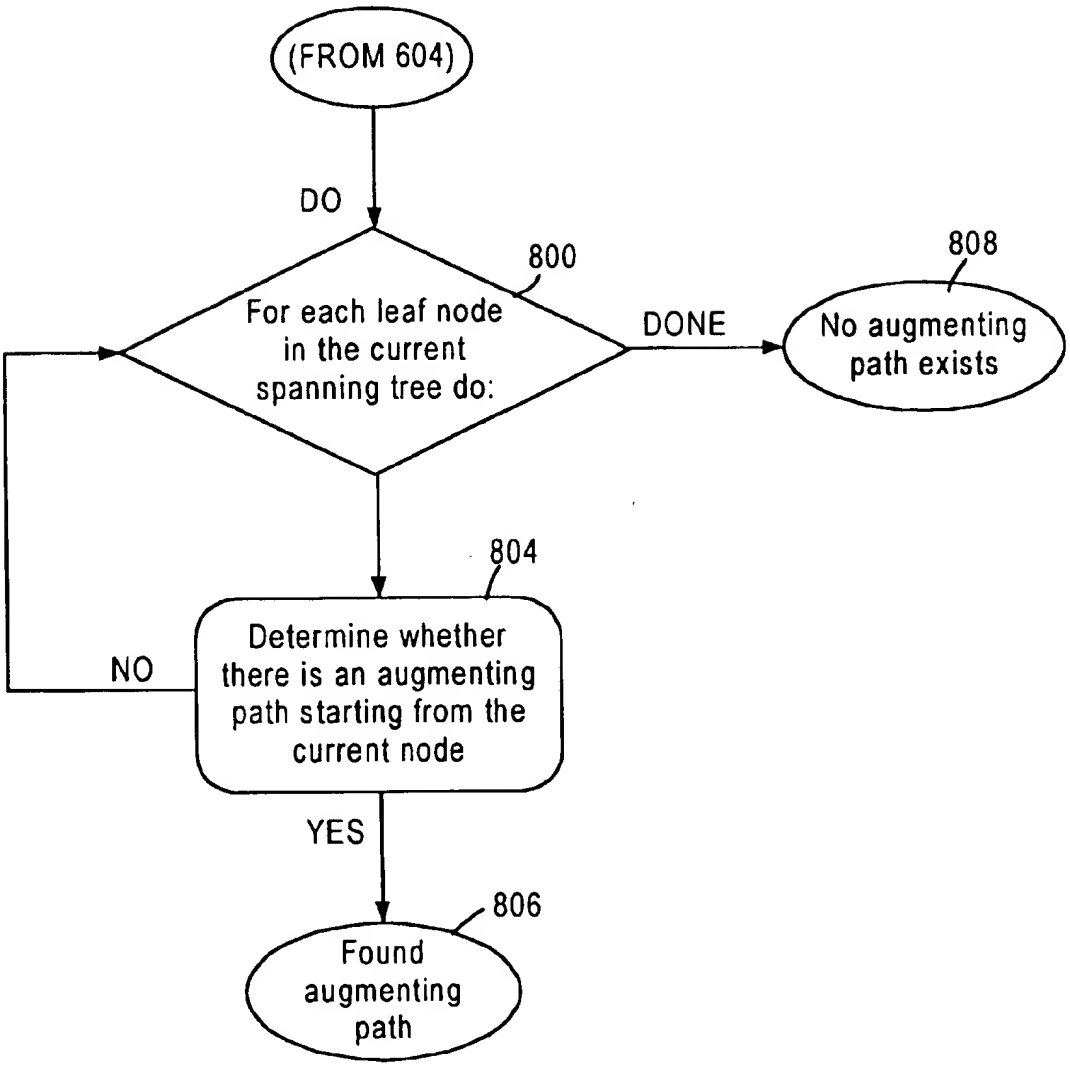

(FROM 604)
DO
800
For each leaf node in the current spanning tree do:
DONE
808
No augmenting path exists
804
Determine whether there is an augmenting path starting from the current node
NO
YES
806
Found augmenting path

FIG. 8(a)

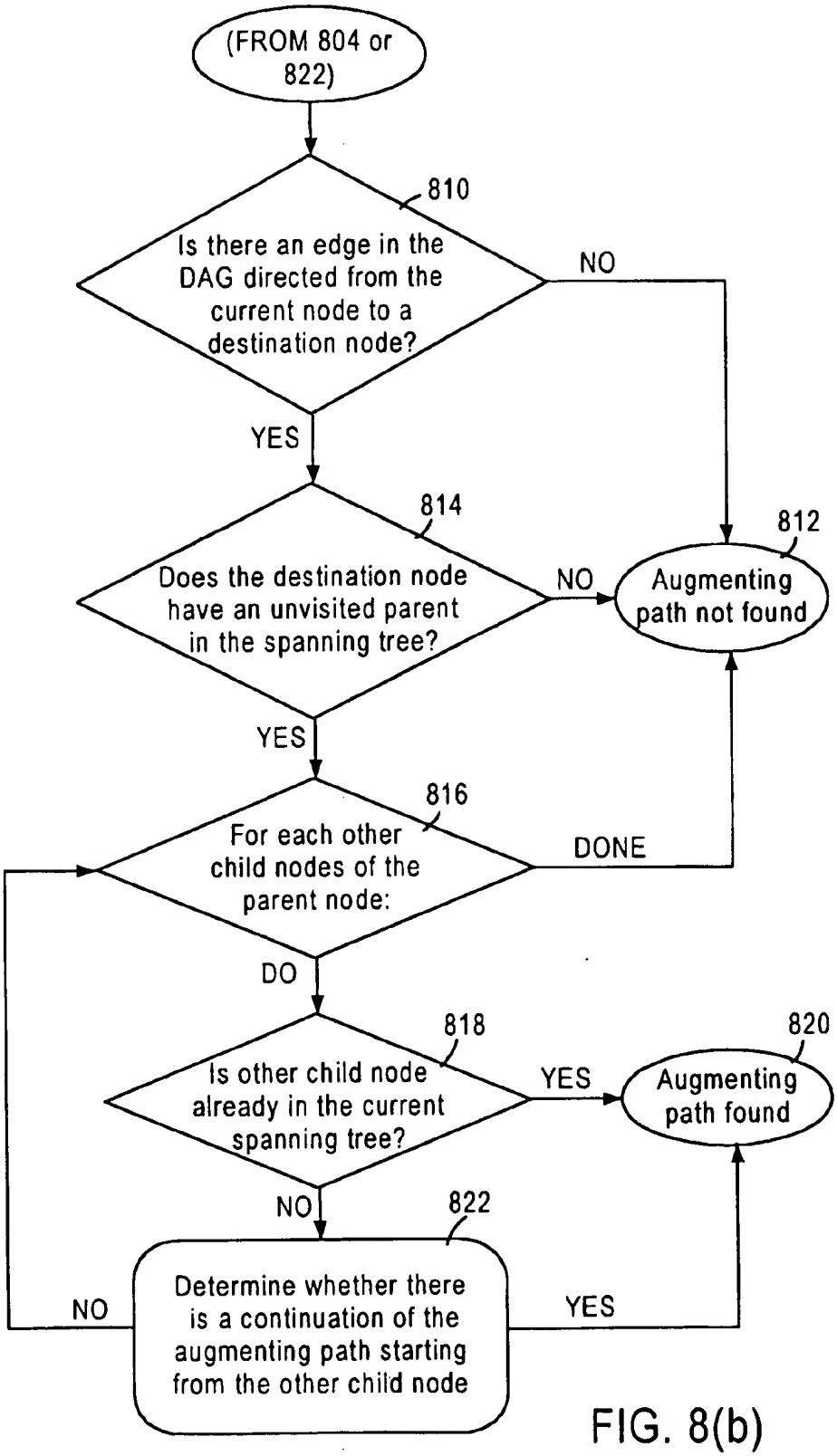

(FROM 804 or 822)
810
Is there an edge in the DAG directed from the current node to a destination node?
NO
YES
814
Does the destination node have an unvisited parent in the spanning tree?
NO
812
Augmenting path not found
YES
816
For each other child nodes of the parent node:
DONE
DO
818
Is other child node already in the current spanning tree?
YES
820
Augmenting path found
NO
822
Determine whether there is a continuation of the augmenting path starting from the other child node
NO
YES

FIG. 8(b)

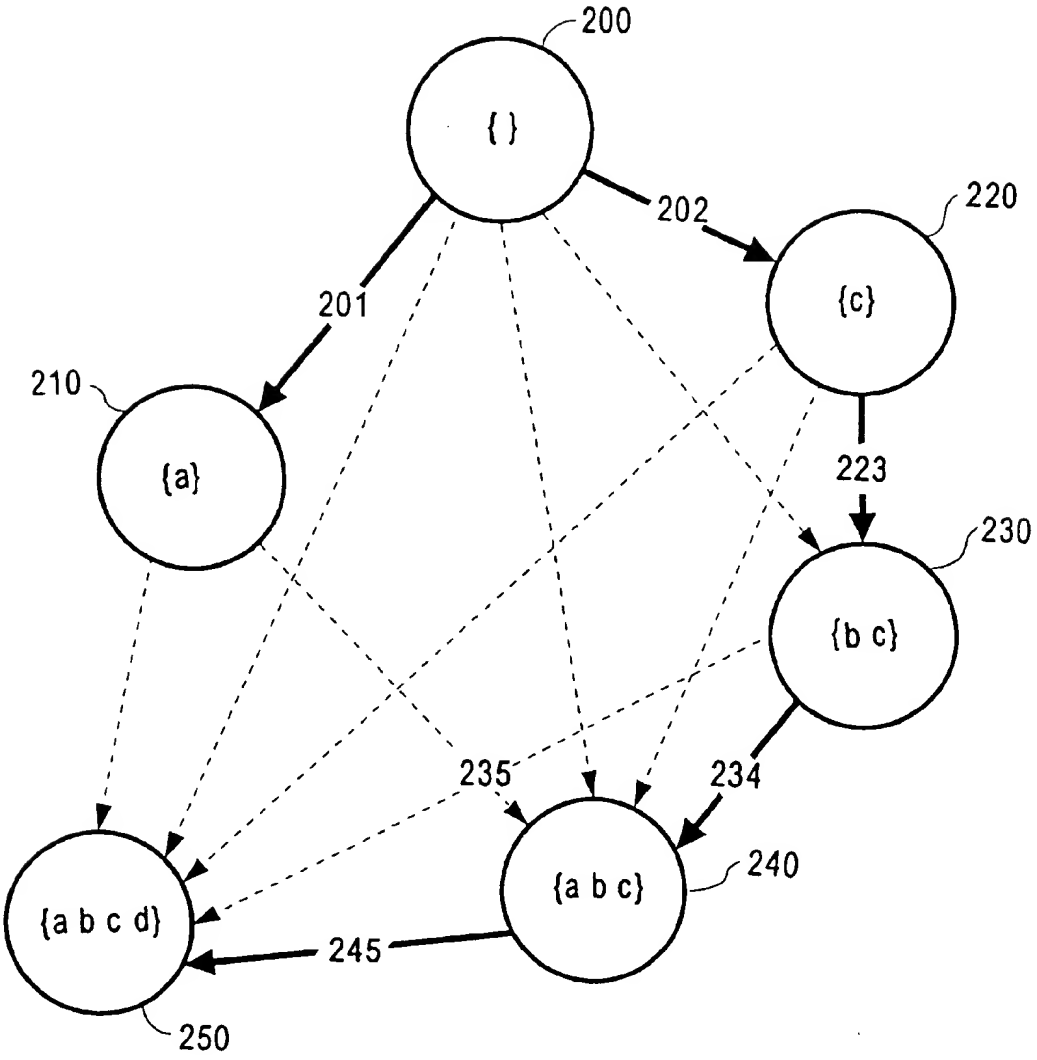



FIG. 9

| | 1010 (ROWID) | 1020 A | 1022 B | 1024 C | 1026 D |
|---|---|---|---|---|---|
| 1030 | 1 | 1 | 1 | 1 | 2 |
| 1032 | 2 | 3 | 7 | 2 | 8 |
| 1034 | 3 | 2 | 4 | 3 | 6 |
| 1036 | 4 | 4 | 2 | 4 | 4 |
| 1038 | 5 | 7 | 8 | 5 | 1 |

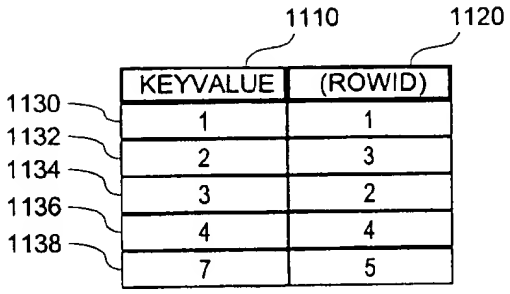
TABLE T1
1000

FIG. 10
(PRIOR ART)

| | KEYVALUE | (ROWID) |
|---|---|---|
| | 1,1 | 1 |
| | 2,7 | 2 |
| 1202 | 3,4 | 3 |
| | 4,2 | 4 |
| | 5,8 | 5 |

INDEX ON C, B
1200

FIG. 12
(PRIOR ART)




INDEX ON A
1100

FIG. 11(a)
(PRIOR ART)

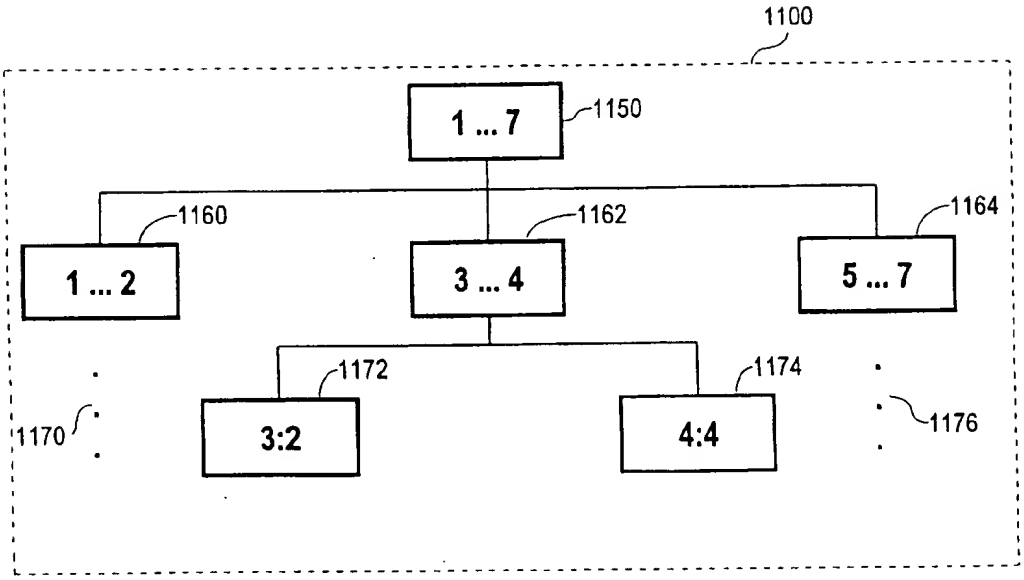



FIG. 11(b)
(PRIOR ART)

# MINIMUM LEAF SPANNING TREE

## FIELD OF THE INVENTION

The present invention relates to computer database systems and more particularly to efficiently executing a query in a database.

## BACKGROUND OF THE INVENTION

In a relational database, information is stored in indexed tables. A user retrieves information from the tables by entering input that is converted to queries by a database application. The database application submits the queries to a database server. In response to a query, the database server accesses the tables specified in the query to determine which information within the tables satisfies the queries. The information that satisfies the queries is then retrieved by the database server and transmitted to the database application and ultimately to the user. Queries may also be internally generated and executed by a database system for performing administrative operations.

For any given database application, the queries must conform to the rules of a particular query language. Most query languages provide users with a variety of ways to specify information to be retrieved. For example, in the Structured Query Language (SQL), the following query requests the retrieval of the information contained in all columns of rows in table T1 in which the value of column a is 2:

Query 1

```
Select * from T1
  where a=2
```

Table T1 (**1000**) is shown in FIG. **10** and comprises four user columns, **1020–1026**, and five rows (**1030–1038**). Table **1000** also has an internal column **1010**, or pseudocolumn, referred to as rowid. A table's rowid pseudocolumn is not displayed when the structure of the table is listed. However, the rowid is retrievable by query and uniquely identifies a row in the table. Rowid pseudocolumn **1010** has rowid entries that correspond to rows **1030–1038**. Thus, a rowid of 2 for table **1000** specifies row **1032** and no other row of table **1000**. Columns **1020–1026** each store data, in this example numbers, and each column has a name. The name of column **1020** is a and the names of columns **1022**, **1024**, and **1026** are b, c, and d, respectively.

Without special processing, a database server would have to fetch every row of a table and inspect every column named in the where clause to perform the query. Such an approach, however, impairs the overall database system performance because many disk blocks would have to be read. Consequently, many database systems provide indexes to increase the speed of the data retrieval process. A database index is similar to a normal index found at the end of a book, in that both kinds of indexes comprise an ordered list of information accompanied with the location of the information. Values in one or more columns are stored in an index, maintained separately from the actual database table.

In FIG. **11**(*a*), index **1100** is an index built on column a of table **1000**. Each entry 113O–**1138** in index **1100** has a key value **1110** and a rowid **1120**. Since the key values are ordered, it can quickly be determined, for example, that the row having a key value of 2 in column a is associated with rowid **3** (see index entry **1132**). An index may be implemented in a variety of ways well known in the art, such as with B-trees, depending on the specific performance characteristics desired for the database system. As changes are made to the table upon which an index is built, the index must be updated to reflect the changes.



FIG. **11**(*b*) shows a B-tree implementation of index **1100**. A B-tree consists of a set of nodes connected in a hierarchical arrangement. A B-tree contains two types of nodes: branch nodes and leaf nodes. Leaf nodes reside at the lowest level of the hierarchy and contain values from the actual column to which the index corresponds. For example, B-tree **1100** is an index for column a **1020** of table **1000** and has leaf nodes **1172** and **1174**. Node **1174** is a leaf node that contains a value from column a **1020**. Along with the values, leaf nodes store the rowid of the rows that contain the values. For example, in addition to the number **3**, leaf node **1172** contains the rowid **2** which corresponds to the row **1032** of table **1000** that contains the number **3** in column **1020**. In other words, leaf node **1172** contains index entry **1134**, and a leaf node may contain more than one index entry.

All the nodes in B-tree **1100** that are not leaf nodes are branch nodes. Branch nodes contain information that indicates a range of values. In the illustrated B-tree **1100**, nodes **1150**, **1160**, **1162**, and **1164** are branch nodes and thus correspond to a range of values. The range of values identified in each branch node is such that all nodes that reside below a given branch node correspond to values that fall within the range of values represented by the branch node. For example, node **1162** is a branch node that corresponds to numbers in the numerical range from three to four. Consequently, nodes **1172** and **1174**, which all reside below node **1162** in the hierarchy, correspond to values that fall within the range from 4 to 6. Reference numbers **1170** and **1176** represent connections to other portions of B-tree **1100** that are not shown.

A database server can use index **1100** to process the exemplary query listed above because index **1100** is built on a column referenced in one of the predicates of the where clause. Specifically, the where clause contains the predicate a=2, and index **1100** is built on column a. Not all indexes built upon a table are useful for executing an arbitrary query. For example, the following query may be executed for table T1 **1000**:

Query 2

```
select * from T1
  where b=2 and c=4
```

In this case, using index **1100**, built upon column a **1020**, does not aid in retrieving data for QUERY 2 because column a is not one of the columns referenced in the where clause. On the other hand, if an index is built upon column b **1022**, column c **1024**, or both, then the performance of data retrieval operations for QUERY 2 can be improved. In particular, a "multi-column index" may be built on more than one column of a table; for example index **1200**, illustrated in FIG. **12**, is built upon columns c **1024** and b **1022**. The key value of a multi-column index is a concatenation of column values from the table upon which the multi-column index was built. For example, in FIG. **12**, the key value for entry **1202** lists a value of 4 taken from column c **1024** of table T1 **1000** followed by a value of 2 from column b **1022** of table **1000**. This key value identifies row **1036** by means of rowid **4**. Thus, multi-column index **1200** can be used in processing QUERY 2, because it was built upon both columns referenced in the query.

One property of a multi-column index is that it improves data processing for "point lookup" queries referencing the first ($n \geqq 1$) columns upon which the multi-column index is built. In contrast to a "range lookup," a point lookup identifies a row or set of rows by specifying a specific value

for one or more columns. Thus, the search criteria associated with point lookups includes an equality "=" operator, but not an inequality operator (e.g. a greater than ">" operator) which identifies a range of rows. In the example, since multi-column index **1200** is built upon column c **1024** and column b **1022** in that order, point lookups referring only to column c **1024** can profitably use multi-column index **1200**. The following QUERY 3 is an example of query that can use a point lookup on multi-column index **1200**:

Query 3

```
select * from T1
  where c=2
```

In certain circumstances, it may be known to a relational database system that there are particular combinations of columns of a table that are most likely to be referenced in queries. For example, it may be known for Table T1 **1000** that QUERIES 1, 2, and 3, are fairly common operations, referencing combinations of columns {a}, {b, }, and {c}, respectively. In addition, it may also be known that combinations of columns {a, b, c} and {a, b, c, d} are commonly used in queries. Conversely, many other combinations of columns are rarely referenced in queries received by the database. In the example, it may be a relatively rare occurrence that only column d is referenced in queries.

Since indexes are useful in improving the processing performance of a relational database, one approach for providing indexes would be to provide an index built on every combination of table columns frequently referenced in queries received by the database. In the example presented above, this approach would call for an index to be created for each of the five frequent combinations of columns, viz. {a}, {c}, {b, c}, {a, b, c} and {a, b, c, d}.

However, building and maintaining an index is costly. For example, each time a row is added to Table T1 **1000**, an entry corresponding to the added row must be added to each index built upon the table. Thus, if there are five indexes built upon table T1 **1000**, then the five indexes have to be updated each time a row is added to the table. Likewise, each index built upon a table must be updated each time a row is deleted from the table or a column value referenced by an index is modified.

Since a query referencing a first combination of columns can use an index built upon a second combination of columns if the first combination is a prefix of the second combination, it is advantageous to use the same index for a query referencing the first combination of columns and for a query referencing the second combination of columns. For example, QUERY 2 and QUERY 3, referencing column combinations {b, c} and {c}, respectively, can use index **1200**, which was built upon columns c and b. Both queries realize the performance benefits of using an index, but the maintenance costs of a second index are eliminated.

Failing to create an index than can handle an anticipated query type results in increased access and retrieval costs of executing the query. On the other hand, creating a separate index to handle each respective anticipated query type may result in excessive index maintenance costs, when it is possible for two queries to share an index.

## SUMMARY OF THE INVENTION

What is needed is a technique for determining a set of indexes for a table that can efficiently handle a group of anticipated query types, each query type referencing a respective combination of the table's columns.

This and other needs are met by the present invention in which an equivalent graph is built based on the combination of the table's columns. A minimum leaf spanning tree for the graph is found and indexes are created for the table based on the minimum leaf spanning tree. The leaves of a spanning tree of the equivalent graph correspond to a set of indexes that can cover the anticipated query types, and minimizing the number of leaves in such a spanning tree results in an efficient set of indexes.

One aspect of the invention is a computer-implemented method and a computer-readable medium bearing instructions arranged to cause one or more processors to perform a method of creating one or more indexes for a body of data arranged in columns, which indexes are used to support query types referencing respective combinations of one or columns. This method comprises the steps of: building a graph based on the respective combinations; finding a minimum leaf spanning tree for the graph; and creating one or more indexes based on the minimum leaf spanning tree.

Another aspect of the invention is a computer-implemented method and a computer-readable medium bearing instructions for finding a minimum leaf spanning tree for a directed acyclic graph (DAG) by finding an initial spanning tree for the DAG and establishing the initial spanning tree as a current spanning tree. If an augmenting path is determined to exist for the current spanning tree, then a new spanning tree having fewer leaves than the current spanning tree is produced based on the augmenting path and established as the current spanning tree. The steps of finding an augmenting path and producing a new spanning tree with a reduced number of leaves are repeatedly performed until an augmenting path can no longer be found. The current spanning tree at the end of this loop is established as the minimum leaf spanning tree.

Additional objects, advantages, and novel features of the present invention will be set forth in part in the description that follows, and in part, will become apparent upon examination or may be learned by practice of the invention. The objects and advantages of the invention may be realized and obtained by means of the instrumentalities and combinations particularly pointed out in the appended claims.

## BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

FIG. **1** depicts a computer system that can be used to implement the present invention;

FIG. **2** depicts a directed, acyclic graph representing prefix relationships for an exemplary combination of columns according to an embodiment of the present invention;

FIG. **3** depicts a spanning tree of the graph shown in FIG. **2**;

FIG. **4** is a flowchart illustrating steps of finding an efficient set of indexes according to an embodiment of the present invention;

FIG. **5** depicts a data structure that can be used to implement the graph and spanning tree shown in FIG. **3**;

FIG. **6** is a flowchart illustrating steps of finding a minimum leaf spanning tree according to an embodiment of the present invention;

FIG. **7**(*a*) illustrates an augmenting path for a spanning tree in a directed acyclic graph;

FIG. **7**(*b*) illustrates a new spanning tree related to the spanning tree and augmenting path shown in FIG. **7**(*a*) that has a fewer number of leaves;

FIGS. 8(*a*) and 8(*b*) are flowcharts illustrating steps of finding an augmenting path for a spanning tree of a graph according to an embodiment of the present invention;

FIG. 9 depicts a minimum leaf spanning tree for the graph shown in FIG. 2;

FIG. 10 depicts an exemplary table;

FIG. 11(*a*) depicts an index built upon the table shown in FIG. 10;

FIG. 11(*b*) illustrates a B-Tree implementation of the index shown in FIG. 11(*a*); and

FIG. 12 depicts a multicolumn index built upon the table shown in FIG. 10.

## DESCRIPTION OF THE PREFERRED EMBODIMENT

A method and apparatus are described for creating one or more indexes for a body of data arranged in columns to support a plurality of query types, each of which referencing a respective combination of one or more of said columns. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form in order to avoid unnecessarily obscuring the present invention.

### Hardware Overview

FIG. 1 is a block diagram that illustrates a computer system **100** upon which an embodiment of the invention may be implemented. Computer system **100** includes a bus **102** or other communication mechanism for communicating information, and a processor **104** coupled with bus **102** for processing information. Computer system **100** also includes a main memory **106**, such as a random access memory (RAM) or other dynamic storage device, coupled to bus **102** for storing information and instructions to be executed by processor **104**. Main memory **106** also may be used for storing temporary variables or other intermediate information during execution of instructions to be executed by processor **104**. Computer system **100** further includes a read only memory (ROM) **108** or other static storage device coupled to bus **102** for storing static information and instructions for processor **104**. A storage device **110**, such as a magnetic disk or optical disk, is provided and coupled to bus **102** for storing information and instructions.

Computer system **100** may be coupled via bus **102** to a display **112**, such as a cathode ray tube (CRT), for displaying information to a computer user. An input device **114**, including alphanumeric and other keys, is coupled to bus **102** for communicating information and command selections to processor **104**. Another type of user input device is cursor control **116**, such as a mouse, a trackball, or cursor direction keys for communicating direction information and command selections to processor **104** and for controlling cursor movement on display **112**. This input device typically has two degrees of freedom in two axes, a first axis (e.g., x) and a second axis (e.g., y), that allows the device to specify positions in a plane.

The invention is related to the use of computer system **100** for creating an efficient set of indexes. According to one embodiment of the invention, creating an efficient set of indexes is provided by computer system **100** in response to processor **104** executing one or more sequences of one or more instructions contained in main memory **106**. Such instructions may be read into main memory **106** from another computer-readable medium, such as storage device **110**. Execution of the sequences of instructions contained in main memory **106** causes processor **104** to perform the process steps described herein. One or more processors in a multi-processing arrangement may also be employed to execute the sequences of instructions contained in main memory **106**. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions to implement the invention. Thus, embodiments of the invention are not limited to any specific combination of hardware circuitry and software.

The term "computer-readable medium" as used herein refers to any medium that participates in providing instructions to processor **104** for execution. Such a medium may take many forms, including but not limited to, non-volatile media, volatile media, and transmission media. Non-volatile media include, for example, optical or magnetic disks, such as storage device **110**. Volatile media include dynamic memory, such as main memory **106**. Transmission media include coaxial cables, copper wire and fiber optics, including the wires that comprise bus **102**. Transmission media can also take the form of acoustic or light waves, such as those generated during radio frequency (RF) and infrared (IR) data communications. Common forms of computer-readable media include, or example, a floppy disk, a flexible disk, hard disk, magnetic tape, any other magnetic medium, a CD-ROM, DVD, any other optical medium, punch cards, paper tape, any other physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read.

Various forms of computer readable media may be involved in carrying one or more sequences of one or more instructions to processor **104** for execution. For example, the instructions may initially be borne on a magnetic disk of a remote computer. The remote computer can load the instructions into its dynamic memory and send the instructions over a telephone line using a modem. A modem local to computer system **100** can receive the data on the telephone line and use an infrared transmitter to convert the data to an infrared signal. An infrared detector coupled to bus **102** can receive the data carried in the infrared signal and place the data on bus **102**. Bus **102** carries the data to main memory **106**, from which processor **104** retrieves and executes the instructions. The instructions received by main memory **106** may optionally be stored on storage device **110** either before or after execution by processor **104**.

Computer system **100** also includes a communication interface **118** coupled to bus **102**. Communication interface **118** provides a two-way data communication coupling to a network link **120** that is connected to a local network **122**. For example, communication interface **118** may be an integrated services digital network (ISDN) card or a modem to provide a data communication connection to a corresponding type of telephone line. As another example, communication interface **118** may be a local area network (LAN) card to provide a data communication connection to a compatible LAN. Wireless links may also be implemented. In any such implementation, communication interface **118** sends and receives electrical, electromagnetic or optical signals that carry digital data streams representing various types of information.

Network link **120** typically provides data communication through one or more networks to other data devices. For

example, network link **120** may provide a connection through local network **122** to a host computer **124** or to data equipment operated by an Internet Service Provider (ISP) **126**. ISP **126** in turn provides data communication services through the world wide packet data communication network, now commonly **395** referred to as the "Internet" **128**. Local network **122** and Internet **128** both use electrical, electromagnetic or optical signals that carry digital data streams. The signals through the various networks and the signals on network link **120** and through communication interface **118**, which carry the digital data to and from computer system **100**, are exemplary forms of carrier waves transporting the information.

Computer system **100** can send messages and receive data, including program code, through the network(s), network link **120**, and communication interface **118**. In the Internet example, a server **130** might transmit a requested code for an application program through Internet **128**, ISP **126**, local network **122** and communication interface **118**. In accordance with the invention, one such downloaded application provides for creating an efficient set of indexes as described herein.

The received code may be executed by processor **104** as it is received, and/or stored in storage device **110**, or other non-volatile storage for later execution. In this manner, computer system **100** may obtain application code in the form of a carrier wave.

### Representing Relationships Between Column Combinations

The relationships between the column combinations referenced by anticipated query types can be expressed in the form of a directed acyclic graph (DAG). A DAG is a data structure comprising nodes connected by edges, in which relationships between nodes are expressed by edges directed from one node in the graph to another node in the graph. The term "acyclic" means that the edges do not form loops in the graph; thus, travelling from node to node in an acyclic graph via directed edges would eventually terminate in a node having no edge emanating therefrom.

Recall that if a first combination of columns is a prefix of a second combination of columns, then a query referencing the first column combination can advantageously use for point lookups an index built upon the columns of the second combination of columns. For example, a query referencing column combination {c} can advantageously use an index built upon columns c and b for point lookups, which is one of the indexes specified by column combination {b, c}. Thus, the column combination {b, c}, of which column combination {c} is a subset, specifies at least one index that column combination {c} can potentially be shared. This relationship may be expressed generally within a DAG by a first node representing a first column combination, a second node representing a second column combination, of which the first column combination is a subset, and an edge directed from the first node to the second node. The exemplary relationship between column combinations {c} and {b, c} can be represented in a DAG by a first node representing column combination {c}, a second node representing column combination {b, c}, and an edge directed from the first node to the second node. As another example, FIG. **2** illustrates a DAG that expresses the relationships between the following combination of columns: {a}, {c}, {b, c}, {a, b, c}, and {a, b, c, d}, corresponding to nodes **210**, **220**, **230**, **240**, and **250**, respectively.

Node **210** represents column combination {a} and has two directed edges **214** and **215** emanating therefrom. Since a query referencing column combination {a} can use an index built upon columns a, b, and c (corresponding to node **240**), there is a directed edge **214** from node **210** to node **240**. Likewise, since a query referencing column combination {a} can use an index built upon columns a, b, c, and d (corresponding to node **250**), there is a directed edge **215** from node **210** to node **250**. There is no edge directed from node **210** to either node **220** or **230**, because a query referencing column combination {a} cannot advantageously use for point lookups an index built upon either column combination {c} or {b, c}, respectively.

Node **220** represents column combination {c} and has three directed edges **223**, **224**, and **225** emanating therefrom. Since a query referencing column combination {c} can use indexes built upon any of column combinations {b, c}, {a, b, c}, and {a, b, c, d}, corresponding to nodes **230**, **240**, and **250**, respectively, the directed edges **223**, **224**, and **225** point to nodes **230**, **240**, and **250**, respectively. There is no edge directed from node **220** to node **210**, because a query referencing column combination {c} cannot advantageously use for point lookups an index built upon column combination {a}.

Node **230** represents column combination {b, c} and has two directed edges **234** and **235** emanating therefrom. Since a query referencing column combination {b, c} can use indexes built upon either of column combinations {a, b, c} and {a, b, c, d}, corresponding to nodes **240**, and **250** respectively, the directed edges **234** and **235** point to nodes **240** and **250**, respectively. There is no edge directed from node **230** to either of nodes **210** or **220**, because a query referencing column combination {b, c} cannot advantageously use for point lookups an index built upon either column combination {a} or {c}, respectively.

Node **240** represents column combination {a, b, c} and has one directed edge **245** emanating therefrom. Since a query referencing column combination {a, b, c} can use an index built upon column combination {a, b, c, d}, corresponding to node **250**, the directed edge **245** points to node **250**. There is no edge directed from node **240** to any of nodes **210**, **220**, and **230**, because a query referencing column combination {a, b, c} cannot advantageously use for point lookups an index built upon any of column combination {a}, {c}, and {b, c}, respectively.

Node **250** represents column combination {a, b, c, d} and has no directed edges emanating therefrom. There is no edge directed from node **250** to any of nodes **210**, **220**, **230**, and **240**, because a query referencing column combination {a, b, c, d} cannot advantageously use for point lookups an index built upon any of column combination {a}, {c}, {b, c}, and {a, b, c} respectively.

Node **200** represents an empty column combination { }, which is trivially a prefix of every other column combination, viz. {a}, {c}, {b, c}, {a, b, c}, and {a, b, c, d}, corresponding to nodes **210**, **220**, **230**, **240**, and **250**, respectively. Accordingly, node **200** has edges **201**, **202**, **203**, **204**, and **205** directed to nodes **210**, **220**, **230**, **240**, and **250**, respectively. However, since none of the other column combinations is a prefix of the empty column combination { }, no other node has an edge directed therefrom to node **200**.

A "rooted DAG" is a DAG that has "a root node" from which every other node is reachable, and, by acyclicity, the root node is unique and has no entering edges. Since node **200** can reach every other node, but no other node can reach node **200**, the addition of node **200** creates a rooted DAG.

### A Spanning Tree of a Graph

A tree is a collection of elements in which one of the elements is designated as a "root" and the remaining

elements, if any, are partitioned into one or more subtrees. Since this definition of a tree is recursive, one of the elements of the subtree is also designated as a root for the subtree and the remaining elements of the subtree, if any, are further partitioned into one or more subtrees. The root of a tree is the "parent" of the root of each constituent subtree; conversely, the root of each subtree is a "child" of the root of the encompassing tree. If a tree (or subtree) consists of only one element, that element is termed a "leaf" element. Thus, a leaf element is not a parent of any other element in the tree.

Elements of a tree can be represented as nodes of a DAG, and the parent-child relationship between the elements can be represented as directed edges in a DAG. Thus, a tree is a kind of a DAG. A spanning tree of a rooted DAG is a tree constructed from the graph using, or "spanning," all the nodes of the graph. Since edges of a DAG are directed, a spanning tree of a root DAG consists of the nodes of the DAG as elements and uses the root node of the DAG as the root of the spanning tree. A spanning tree typically has fewer edges than the graph it spans. For example, a node in a DAG may be the destination of two or more edges, but only one of those edges would be in any one spanning tree of the graph. In general, a DAG can have more edges than nodes, but a spanning tree has exactly one fewer edge than nodes. For example, the DAG depicted in FIG. **2** has thirteen edges for six nodes, but spanning trees of the DAG contain only five edges.

FIG. **3** depicts an exemplary spanning tree for the DAG illustrated in FIG. **2**, which represents the relationships between the exemplary combinations of columns. The edges of the DAG that belong to the exemplary spanning tree are depicted as solid arrows. For example, edge **201** from node **200** to node **210** is shown as a solid arrow and belongs to the exemplary spanning tree. The other edges in the spanning tree include edge **202** from node **200** to node **220**, edge **223** from node **220** to node **230**, edge **234** from node **230** to node **240**, and edge **235** from node **230** to node **250**. Referring again to FIG. **3**, edges belonging to the DAG that are not in the spanning tree are depicted by a dashed arrow. For example, edge **245** from node **240** to node **250** is shown by a dashed arrow and is not in the exemplary spanning tree. There are five edges in the spanning tree for six nodes of the DAG.

A leaf of a spanning tree is not a parent of any other node in the spanning tree. In other words, a leaf node does not have any edges directed from itself in the spanning tree. Referring again to FIG. **3**, node **250** is a leaf node in the exemplary spanning tree because there are no edges emanating therefrom. Node **240** is also a leaf node of the exemplary spanning tree, because the only edge emanating therefrom, namely edge **245** from node **240** to node **250**, is not in the exemplary spanning tree. Node **220**, however, is not a leaf node of the exemplary spanning tree, because node **220** has an edge **223** from node **220** to node **230** that is in the exemplary spanning tree and depicted with a solid arrow.

Conventional techniques such as a depth-first search or breadth-first search exist for finding a spanning tree for a DAG. A depth-first search is typically implemented by a recursive subroutine in which edges are successively followed from node to node until a leaf node is reached. When a leaf node is reached, the depth-first search backs up and checks previous nodes for additional edges to as-yet-unvisited nodes to add to the initial spanning tree.

For example, referring back to FIG. **2**, root node **200** has five exiting edges **201**, **202**, **203**, **204**, and **205**. Among the edges **201**, **202**, **203**, **204**, and **205**, a depth-first search may chose and traverse edge **202** to reach node **220**, which is the source for nodes **223**, **224**, and **225**. Subsequently, the depth-first search may traverse edge **223** to reach node **230**. At node **230**, edge **235** may be traversed to reach node **250**, which lacks an edge emanating therefrom. Accordingly, the depth-first search backs up a level to node **230** and selects a remaining edge, namely **234**, to traverse, reaching node **240**. Although node **240** has an edge **245** emanating to node **250**, node **250** has already been visited by the depth-first search, so that the depth-first search returns to node **230**. Since all destination nodes from node **230**, viz. nodes **240** and **250**, have also been visited, the depth-first search returns back to node **220** and thence to root node **200**. At this point, the depth-first search traverses edge **201** to reach node **210**, since edges **203**, **204**, and **205** point to visited nodes **230**, **240**, and **250**, respectively.

This exemplary depth-first search finds an initial spanning tree comprising edges **201**, **202**, **223**, **234**, and **235** and illustrated in FIG. **3**. The particular spanning tree found by a depth-first search is typically dependent on the particular order in which edges from a node are consulted or stored in a data structure.

### Correspondence Between Spanning Tree Leaves and Indexes

The leaves of a spanning tree of a graph representing subset relationships between combinations of columns correspond to the set of indexes that can cover all the anticipated queries. For example, the spanning tree depicted in FIG. **3** has three leaves: node **210** representing column combination {a}, node **240** representing column combination {a, b, c}, and node **250** representing column combination {a, b, c, d}. Accordingly, the exemplary spanning tree indicates that three indexes may be built upon the corresponding column combinations in order to support all the exemplary anticipated query types. In the example, since node **210** representing column combination {a} is a leaf node in the spanning tree, the spanning tree indicates that an index may be built upon column a.

For indexes built on a plurality of columns, i.e., multi-column indexes, the order of columns is significant. More specifically, a multi-column index is built having the prefixed columns placed before non-prefixed columns. In other words, those columns specified in the ancestor nodes of a leaf node in the spanning tree come before those columns specified only in the leaf node. In the example, leaf node **240** represents column combination {a, b, c} and has node **230** representing column combination {b, c} as a parent and thence node **220** representing column combination {c}. Thus, the multi-column index corresponding to node **240** is built on column c (specified in node **220**), column b (specified in node **230**), and then column a (specified in node **240**). Leaf node **250** includes two columns not specified in any ultimate parent node, viz. columns a and column d. In this case, the order of non-prefixed columns is immaterial for building an index; thus, either an index on columns c, b, a, and d or on columns c, b, d, and a may be built.

In the example, the three indexes, a first index built on column a, a second index built on columns c, b, and a, and a third index built on columns c, b, d, and a, can cover the exemplary anticipated query types referencing column combinations {a}, {c}, {b, c}, {a, b, c}, and {a, b, c, d}. More specifically, a query referencing column combination {a} can use the first index built on column a. The queries referencing column combinations {c} and {b, c} can use

either the second index built on columns c, b, and a, or the third index built on columns c, b, d, and a. The query referencing column combination {a, b, c} can use the second index built on columns c, b, and a. Finally, the query referencing column combination {a, b, c, d} can use the third index built on columns c, b, d, and a.

### A Minimum Leaf Spanning Tree

A minimum leaf spanning tree of a graph is a spanning tree of the graph such that no other spanning tree of the graph has a fewer number of leaves than the minimum leaf spanning tree. In other words, the minimum leaf spanning tree of the graph has the fewest possible, or minimum, number of leaves. Moreover, a plurality of minimum leaf spanning trees can exist for a given DAG.

A minimum leaf spanning tree differs conceptually from a conventional "minimum spanning tree," which minimizes the aggregate weight of edges within a graph. For clarity, such conventional minimum spanning trees are termed herein as "minimum edge-weight spanning trees." Since all spanning trees of a graph have the same number of edges, one less than the number of nodes, minimum edge-weight spanning trees are most meaningful for graphs that have weighted edges. There are a variety of well-known techniques for finding minimum edge-weight spanning trees, for example, Kruskal's algorithm, Prim's algorithm, and Boruvka's algorithm, none of which, however, are designed to find a minimum leafspanning tree. A minimum leaf spanning tree, on the other hand, is a spanning tree that has a minimal number of leaves without consideration of weights of the edges. Thus, a minimum leaf spanning tree is well defined even for graph whose edges are not assigned weights.

As mentioned hereinabove, there is a need for determining a minimal set of indexes for a table that can efficiently handle a group of anticipated query types, each query type referencing a respective combination of the table's columns. Since the leaves of a spanning tree of a graph representing the relationships between the column combinations indicate indexes that can cover all the anticipated query types and since for any set of n indexes there is a spanning tree having at most n leaves, the leaves of a minimal spanning tree indicate a minimal set of indexes that can efficiently handle the anticipated query types. Accordingly, one embodiment of the present invention meets this need by performing the steps illustrated in FIG. **4**.

Referring to FIG. **4**, a directed acyclic graph (DAG) equivalent to the anticipated column combinations (step **400**) is built. A minimum leaf spanning tree is found for the DAG (step **402**). A group of indexes is then built based on column combinations associated with the leaves of the minimum leaf spanning tree (step **406**). Each of these steps shall be described in greater detail hereinafter.

### Building an Equivalent Graph

Referring to step **400** in FIG. **4**, a directed, acyclic graph (DAG) equivalent to the pattern of anticipated query types is built. In particular, the nodes of the DAG correspond to the respective column combinations, and the directed edges correspond to a subset relationship existing between column combinations. Moreover, the DAG is built with a root node that can reach every other node in the DAG. More formally, the nodes of such a DAG are { } and each column combination $n_i$, and the edges of the DAG are { }$\rightarrow n_j$ for all j and $n_{i \rightarrow nj}$ if and only if $n_i \subseteq n_j$. For the working example of anticipated query types referencing column combinations {a}, {c}, {b, c}, {a, b, c}, and {a, b, c, d}, such an equivalent DAG is illustrated in FIG. **2**, as described in more detail hereinbelow.

There is a variety of techniques and data structures for implementing a directed, acyclic graph, but the present invention is not limited to any particular technique or data structure. An "object-based" approach defines an object (e.g. a structure, record, instance of an abstract data type, or other equivalent construct depending on the programming language) to hold information for each vertex. Edges in an object-based approach are implemented by another object or equivalent construct, which includes a reference (e.g pointers, cursors, indexes, addresses, and the like), to the vertices they connect. In an adjacency list implementation, the edges that come from a vertex are implemented as a collection of references to the respective vertices the edges connect. An incidence list combines the object-based approach and the adjacency list approach, in which each vertex object includes a linked list of edge objects pointing to vertices.

An incident list representation is depicted in FIG. **5** for the DAG in the working example. Vertex object **500** contains data for vertex **200** and may include the following fields: an optional NODE field to hold an identifier of the vertex (**200**), a COL field for the column combination represented by the node, a linked list EDGES of edge objects **501**, **502**, **503**, **504**, and **505**, and PARENT field to indicate the parent node in a spanning tree for the graph. As described in more detail herein below, a MARK field is used to keep track of whether the node has been "visited" within a pass. Each edge object **501–505** contains a reference to another vertex and a link to the next edge in the list. For example, the reference in edge object **501** is "210" indicating vertex **210** of the graph. In FIG. **5**, the reference is a value or "cursor" of the identifier for the associated vertex, however, other implementations may employ a pointer to the associated vertex object, such as a virtual address of the start of the associated vertex object.

Referring again to FIG. **5**, vertex object **510** represents vertex **210** of the graph (NODE field) and has a linked list of edge objects containing edge objects **514** and **515**, which refer to vertex objects **540** and **550**, respectively. Vertex object **520** represents vertex **220** of the graph (NODE field) and has a linked list of edge objects containing edge objects **523**, **524** and **525**, which refer to vertex objects **530**, **540** and **550**, respectively. Vertex object **530** represents vertex **230** of the graph (NODE field) and has a linked list of edge objects containing edge objects **534** and **535**, which refer to vertex objects **540** and **550**, respectively. Vertex object **540** represents vertex **240** of the graph NODE field) and has a linked list of edge objects containing edge object **545**, which refers to vertex object **550**. Finally, vertex object **550** represent vertex **250** of the graph (NODE field) and has a null linked list of edge objects.

Other approaches include an "adjacency matrix" in which cells of a square matrix having rows and columns indexed by vertices indicate whether the vertices for the row and column are connected. Another matrix is an "incidence matrix" has rows indexed by vertices and columns indexed by edges, in which each cell in the matrix indicates whether the vertex and the edge are incident. Other techniques can be used to implement a DAG equivalent to the pattern of column combinations referenced by anticipated query types.

A spanning tree of a graph can be implemented by a separate data structure that includes pointers to the vertex objects or, preferably, within the same data structure that implements the graph and reusing the vertex objects. Since a root node in a spanning tree can have a plurality of subtrees, the subtrees of a root node can be represented by a second linked list of child edge objects. Another approach

is to include an additional flag in each edge object of the associated linked list of edge objects, wherein the flag indicates whether the edge represented by the edge object is in the spanning tree. Since each node in a spanning tree can have at most one parent node, yet another approach includes an extra field in each vertex object to indicate the parent node in the spanning tree. In the data structure illustrated in FIG. **5**, the PARENT field of vertex object **550** includes a reference **230** (or an equivalent such as a pointer) to its parent, vertex **230**, in the spanning tree depicted in FIG. **3**. The PARENT field of vertex object **540** indicates vertex **230** as the parent, and the PARENT fields of vertex objects **510**, **520**, and **530** indicate vertices **200**, **200**, and **220**, respectively, as the parent. The PARENT field of vertex object **500** is null, since vertex **200** is the root of the spanning tree.

Finding a Minimum Leaf Spanning Tree

After the equivalent DAG is constructed, an embodiment of the present invention finds a minimum leaf spanning tree of the DAG (step **402**). Although a plurality of minimum leaf spanning trees may exist for a DAG, only one minimum leaf spanning tree need be found to determine a minimal set of indexes for a given set of anticipated query types. On the other hand, it is contemplated that other embodiments of the present invention are configured to find two or more of minimum leaf spanning trees of a DAG and choose one of them based on ranking criteria. For example, the net cost for using indexes indicated by leaves of a minimum leaf spanning tree can be calculated by computing selectivity factors for the anticipated queries multiplied by a cost metric for each index as disclosed in the commonly assigned, U.S. application Ser. No. 08/808,094 entitled "Index Selection for an Index Access Path" and filed on Feb. 28, 1997 by Hakan Jakobsson, Michael Depledge, Cetin Ozbutin, and Jeffrey I. Cohen (now U.S. Pat. No. 5,924,088), incorporated herein by reference.

One method of finding a minimum leaf spanning tree is illustrated in FIG. **6**. In step **600**, in which an initial spanning tree is found for the DAG, as by conventional techniques such as a depth-first search and a breadth-first search, as described in more detail herein above. The present invention is not limited to any particular initial spanning tree or to any particular method of finding an initial spanning tree, which may vary from implementation to implementation. In the working example, one initial spanning tree is illustrated in FIG. **3** and has three leaves: node **210**, node **240**, and node **250**.

In step **602**, the initial spanning tree is established as the current spanning tree for a loop that repeatedly finds related spanning trees with fewer leaf nodes until no more are found. According to one embodiment of the present invention, such a related spanning tree is determined by finding an augmenting path for the current spanning tree in the DAG, if it exists (steps **604** and **606**) and adjusting the edges of the spanning tree based on the augmenting path to produce a spanning tree with a reduced number of leaves (step **608**).

Finding an Augmenting Path

In step **604**, an augmenting path for the current spanning tree of the DAG is found, if it exists. One exposition of an augmenting path in a different context, viz. maximal matchings in a bipartite graph, is found in Aho, Hopcroft & Ullman, *Data Structure & Algorithms* (Reading, Mass.: Addison-Wesley, 1983). An augmenting path may be formally defined as follows: given a spanning tree T of a graph G=(V, E), an augmenting path for T is a sequence $v_0, v_1, \ldots, v_k$ of vertices from V such that:

(1) $v_0$ is a leaf of T;

(2) for $0<j<k$ each of $v_j$ has exactly one child in T;

(3) $v_k$ has at least two children in T; and

(4) for $0 \leqq i<k$, there exists a vertex $u_i$ such that (a) the edge $v_i \rightarrow u_i$ is in E but not in T and (b) the edge $v_{i+1} \rightarrow u_i$ is in T.

This formal definition can be visualized with reference to FIG. **7**(*a*), which depicts a portion of a directed, acyclic graph comprising nodes $v_0, v_1, v_2, \ldots V_k, u_0, u_1, \ldots, u_{k-1}$, and w. The edges represented by solid arrows are in the spanning tree and the graph, and the edges represented by dashed arrows are in the graph but not in the spanning tree. In FIG. **7**(*a*), node $v_0$ is a leaf node, condition (1), and nodes $v_1, v_2, \ldots, v_k$ are non-leaf nodes. Each of nodes $u_0, u_1, \ldots, u_{k-1}$, and w can be either a leaf node or a non-leaf node. Considering condition (2), each of nodes $v_1, v_2, \ldots, v_{k-1}$, has only one child in the spanning tree, indicated by a solid arrow. For condition (3), node $v_k$ has two children, node $u_{k-1}$ and node w. For each of nodes $u_0, u_1, \ldots, u_{k-1}$, the parent node to the left, marked by a dashed arrow, is not in the spanning tree, and the parent node to the right is in the spanning tree, meeting conditions (4a) and (4b) respectively.

One augmenting path for the spanning tree in the working example of FIG. **3** comprises $v_0$ as node **240** and $v_1$ as node **230**, where k=1 and $u_0$ is node **250**. Referring to the definition, $v_0$ (node **240**) is a leaf of T, since node **240** does not have an edge in the spanning tree emanating therefrom. Condition (2) is trivially satisfied since k=1. The third condition is met since $v_1$ (node **230**) has two children in the spanning tree: node **240** via edge **234** and node **250** via edge **235**. Concerning condition (4), edge $v_0 \rightarrow u_0$ (edge **245** from node **240** to node **250**) is not in the spanning tree, but edge $v_1 \rightarrow u_0$ (edge **235** from node **230** to node **250**) is in the spanning tree.

FIGS. **8**(*a*) and **8**(*b*) are flowcharts illustrating one method for finding an augmenting path for a spanning tree of a DAG, if it exists. Step **800** controls a loop that iterates through each leaf node in the current spanning tree until an augmenting path is found. The iteration can be performed by such techniques as a pre-order traversal of the spanning tree. If all the leaf nodes have been exhausted without finding an augmenting path (see step **806**), then the loop controlled by step **800** terminates and execution passes to step **802**, where the lack of an augmenting path is signaled, as by returning a "false" boolean value or equivalent.

During the execution of the loop at step **804**, each leaf node under consideration is established as the current node for finding an augmenting path starting from the current node. Referring back to the definition of an augmenting path, the requirement that the start of the augmenting path be a leaf node satisfies condition (1) that $v_0$ is a leaf of T. During step **804**, the DAG is search for an augmenting path that starts at the current node. In one implementation, this search is performed in a separate subroutine (e.g. a C function) whose operation is illustrated in FIG. **8**(*b*) starting at step **810**. If the result of searching for an augmenting path in step **804** is true, then the method indicated that an augmenting path has been found (step **806**). Otherwise, execution loops back to step **800** where another leaf node, if available, is considered.

Referring to FIG. **8**(*b*), in step **810**, the current node is checked for an edge in the DAG that is directed from the current node. Referring back to the working example depicted in FIG. **3**, leaf node **250** does not have such an

edge; consequently, the "NO" branch is taken, indicating that an augmenting path is not found for the current node (step **812**). If step **810** to find a particular augmenting path was called from step **804** in the main loop, then returning a not found indication causes another iteration of the loop controlled by step **800** for another leaf node, if present. With respect to leaf node **240**, there is an edge directed therefrom: edge **245** directed to node **250**. Accordingly, execution proceeds to step **814**. Since the current node is a leaf node, any edge directed therefrom is in the DAG but not in the spanning tree. Therefore, the test in step **810** checks for condition (4a) that edge $v_i \rightarrow u_i$ is in E but not in T. In the working example, edge $v_0 \rightarrow u_0$ is edge **245**, which is directed from node **240** as $v_0$ to node **250** as $u_0$, is not in the current spanning tree.

At step **814**, the destination node of the edge is checked to determine whether its parent node in the spanning tree has been visited. In the working example, the parent of destination node **250** in the spanning tree is node **230**, since edge **235** is in the spanning tree. On the other hand, node **200**, the root of the DAG and the spanning tree, does not have a parent and, consequently, does not meet this condition Referring again to FIG. **5**, it is evident that the parent of a node can be readily determined according to one embodiment of the present invention by accessing the PARENT field. More specifically, the value of the PARENT field in vertex object **550** representing node **250** refers to node **230**. By finding a parent node in the spanning tree for the destination node, condition (4b) that edge $v_{i+1} \rightarrow u_i$ is in T is satisfied, since edge **235** from node **230** as $v_1$ to node **250** as $u_0$ is part of the current spanning tree.

There are at least two advantages for checking whether the parent node has been visited. One benefit is avoiding infinite loops, and another benefit is the elimination of superfluous attempts to find an augmenting path for nodes already determined not to contain an augmenting path. There is a variety of techniques for determining whether a node has been visited, but the present invention is not limited to any particular technique. For example, a separate data structure can be maintained to record which nodes have been visited. As another example, the data structure that represent vertices in the DAG can be augmented to include a MARK field to hold a Boolean flag that marks whether the corresponding node has been visited. A drawback of the Boolean flag approach is that the data structure for the DAG must be traversed each time to reset the flag for each separate pass of finding an augmenting path. Accordingly, the MARK field preferably contains a monotonically increasing (or, alternatively, decreasing) pass number that indicates the last pass in which the node was visited. Thus, determining whether a node has been visited is performed by comparing the MARK field of the node to the current pass number. If the condition in step **814** is not met, then an indication that an augmenting path is not found for the current node made (step **812**); otherwise, execution proceeds to step **816**. A node is considered and marked as visited when the condition in the next step **816** is met.

In step **816**, the parent node of the destination node is checked for the existence of one or more other child nodes. If there is no other child node, then neither condition (3) that $v_k$ has at least two children in T nor condition (4a) that edge $v_i \rightarrow u_i$ is in E but not in T can be satisfied. Accordingly, execution proceeds to step **812** to indicate that an augmenting path is not present for the current node. In the working example, however, there is another child node for parent node **230**, namely node **240** via edge **234**.

If the edge to any other child node of the parent node is in the current spanning tree, checked by step **818**, then condition (3) that $v_k$ has at least two children in T is satisfied and an augmenting path has been found. In the working example, since $v_1$ as node **230** has at least two children in the current spanning tree, namely node **240** and node **250**, an augmenting path has been found comprising $v_0$ as node **240** and $v_1$ as node **230**, where k=1 and $u_0$ is node **250**. Accordingly, execution branches to step **820**, where the fact that an augmenting path is found is signaled, as by returning a "true" boolean value or equivalent.

On the other hand, if the edges to the other child nodes of the parent node are not in the current spanning tree, condition (4a) that edge $v_i \rightarrow u_i$ is in E but not in T is satisfied. Condition (2) that each of $v_i$ has exactly one child in T is also satisfied, since this parent node has one child in the current spanning (determined in step **814**) but no other child in the current spanning tree (determined in step **818**). Consequently, the search for an augmenting path is continued using one of the other child nodes as the current node (step **822**). One approach is preferably a recursive call to step **810**, as a depth-first search, but other equivalent approaches, such as a search with an explicit stack or other supplementary data structure, may be employed. If the result of searching for a continuation of the augmenting path succeeds, then execution branches to step **820** where this success is signaled. On the other hand, if the result of searching for a continuation of the augmenting path does not succeed, then execution backtracks to step **816** to examine another child node, if it exists, for a potential continuation of the augmenting path. If no other child node exists, then execution reaches step **812** indicating that an augmenting path cannot be continued from the current node.

Reducing a Spanning Tree

Referring back to FIG. **6**, execution proceeds to step **606**, where the existence of an augmenting path possibly found in step **604** is tested. Preferably, step **604** is coded as a routine configured to perform the steps illustrated in FIG. **8** and return a value, e.g a Boolean, indicating whether an augmenting path was found by the routine. If an augmenting path was found, then execution branches to step **608**, wherein the edges of the current spanning tree are adjusted based on the augmenting path to produce a new spanning tree having a fewer number of leaves. On the other hand, if no augmenting path exists, then the current spanning tree is established as a minimum spanning tree (step **610**) and execution returns back to step **404**.

Given a spanning tree of a graph and an augmenting path for the spanning tree, a new spanning tree can be constructed based thereon having a fewer number of leaves. Referring back to FIG. **7**(*a*), such a reduced leaf spanning tree is constructed by deleting all edges $v_{i+1} \rightarrow u_i$ in the augmenting path and replacing them with edges $v_i \rightarrow u_i$. Thus, the parent of $u_i$ in the spanning tree changes from $v_{i+1}$ to $v_i$. A result of this procedure is depicted in FIG. **7**(*b*). By inspection, the new spanning tree has one fewer leaf than the original spanning tree, because $v_0$ changes from a leaf to a non-leaf, $v_1, v_2, \ldots, v_k$ remain non-leaf nodes, and the leaf-ness of nodes $u_0, u_1, \ldots, u_{k-1}$, and w are unaffected.

In the working example, one augmenting path was found comprising $v_0$ as leaf node **240** and $v_1$ as node **230**, where k=1 and $u_0$ is node **250**. Edge **245** from node **240** ($v_0$) to node **250** ($u_0$), marked as a dashed arrow, is not in the spanning tree, and edge **235** from node **230** ($v_1$) to node **250** ($u_0$), marked as a solid arrow, is in spanning tree. Accordingly, a reduced leaf spanning tree is constructed by removing edge **235** from node **230** ($v_1$) to node **250** ($u_0$) from the spanning tree and adding edge **245** from node **240**

($v_0$) to node **250** ($u_0$) into the spanning tree. This reduced leaf spanning tree is illustrated in FIG. **9** and consists of only two nodes **210** and **250** are leaf nodes in the spanning tree, whereas the spanning tree depicted in FIG. **3** comprises three nodes. Edge **235** from node **230** ($v_1$) to node **250** ($u_0$) is marked with a dashed arrow and is not in the spanning tree, and edge **245** from node **240** ($v_0$) to node **250** ($u_0$) is marked with a solid arrow and is in the spanning tree. The reduction of the current spanning tree based on the augmenting path can be performed by a separate subroutine or integrated with the routine that found the augmenting path. In the latter case, the edges can be flipped in or out of the current spanning tree at step **812** by resetting the PARENT field of the vertex object representing node $u_i$ to reference node $v_i$.

With reference to FIG. **6**, after adjusting the edges of a current spanning tree to produced a reduced leaf spanning tree in step **608**, the reduced leaf spanning tree is established as the new current spanning tree and steps **604**, **606**, and **608** are repeated until an augmenting path can no longer be found. In this situation, the loop terminated and the current spanning tree is returned to step **404** as the minimum leaf spanning tree.

### Building the Indexes from the Minimum Leaf Spanning Tree

As described herein above, the leaves of a spanning tree of a graph representing prefix relationships between combinations of columns correspond to a set of indexes that cover all the anticipated queries. Referring back to FIG. **4**, in step **404** the indexes to be built are determined from the minimum leaf spanning tree generated within step **402**. Since a minimum leaf spanning tree has a minimum number of leaves and since for any set of n indexes there is a spanning tree having at most n leaves, determining the set of indexes to be built from the minimum leaf spanning tree results in a minimum number of indexes being built, thereby reducing the costs of maintaining indexes while still efficiently handling the anticipated query types. In the working example, the minimum leaf spanning tree depicted in FIG. **9** has two leaves: node **210** representing column combination {a} and node **250** representing column combination {a, b, c, d}. Accordingly, the minimum leaf spanning tree indicates that two indexes may be built upon the corresponding column combinations in order to support all the exemplary anticipated query types. Since node **210** representing column combination {a} is a leaf node in the spanning tree, the minimum leaf spanning tree indicates that an index may be built upon column a.

Since the order of columns in a multi-column index is significant, the column combinations that form prefixes of other column combination must occur before the non-prefixed columns. Specifically, the column combinations are built in reverse order of the column combinations represented by nodes on the path from a leaf node to the root. According to one embodiment of the present invention, this ordering may be determined by traversing via the PARENT field the minimum leaf spanning from a leaf node to the root while building a stack of column combinations. Due to the LIFO (last in, first out) nature of a stack, pulling column combinations from the stack results in a proper, reverse order of column combinations. The stack need not be explicit, as described herein, because a series of recursive function calls achieve a similar result by an implicit use of a call stack. Other approaches such as lists and queues may also be adopted.

In the working example, starting from multi-column leaf node **250**, column combination {a, b, c, d} is pushed on the stack, and parent node **240** is visited. At node **240**, column combination {a, b, c} is pushed onto the stack so that the stack contains the following elements: ({a, b, c}, {a, b, c, d}). Subsequently, at next parent node **230**, column combination {b, c} is pushed onto the stack so that the stack contains the following elements: ({b, c}, {a, b, c}, {a, b, c, d}). Subsequently, at next parent node **220**, column combination {c} is pushed onto the stack so that the stack contains the following elements: ({c}, {b, c}, {a, b, c}, {a, b, c, d}). Since the parent of node **220** is the root node, the order of columns can be determined by pulling of the top elements of the stack. First, column combination {c} is pulled off the stack; thus the first column in the multi-column index is column c. The next column combination to be pulled off the stack is column combination {b, c} and the new column b is placed after column c, resulting in the order: c and b. Next, column combination {a, b, c} is pulled off the stack and new column a is added to the order of columns, resulting in the order: c, b, and a. Finally, column combination {a, b, c, d} is pulled off the stack and new column d is added to the order of columns, resulting in c, b, a, and d.

Therefore, the two indexes of the working example, a first index built on column a, and a second index built on columns c, b, a, and d, can cover the exemplary anticipated query types referencing column combinations {a}, {c}, {b, c}, {a, b, c}, and {a, b, c, d}. More specifically, a query referencing column combination {a} can use the first index built on column a. The query type referencing column combinations {c} can use the multi-column index built upon columns c, b, a, and d, because column c appears first. A query referencing columns b and c can use the multi-column index built on columns c, b, a, and d because columns c and b appear first. Similarly, the query referencing column combination {a, b, c} and the query referencing column combination {a, b, c, d} can use the multi-column index built on columns c, b, a, and d.

### Subquery Snapshots

The present application may be applied to improve the efficiency of fast refresh of snapshots defined by a query containing a subquery. A snapshot is a body of data constructed of data from a "master" table. The data contained within a snapshot is defined by a query that references the master table and optionally other tables, views, or snapshots. A snapshot can be refreshed periodically or on demand by a user to reflect the current state of its corresponding base tables.

One method of refreshing a table is called "fast refresh," which transfers to the snapshot only those changes to the master table that have been made since the last refresh of the snapshot. A "master log" file can be employed to track and record the rows that have been updated in the master table. When a snapshot is refreshed, only the appropriate rows in the master log need to be applied to the snapshot table. In a networked environment, only those modified rows found at the master site are transferred across the network and updated or inserted into the snapshot. Rows deleted in the master table are also deleted in the snapshot. Fast refresh is typically faster, more efficient, and involves less network traffic than another form of refresh, called "complete refresh," in which the snapshot definition query is merely reissued.

As described in more detail in the commonly assigned U.S. application Ser. No. 08/880,928, entitled "Fast Refresh of Snapshots Containing Subqueries," filed on Jun. 23, 1997 by Alan Downing, Harry Sun, and Ashish Gupta, the con-

tents of which are incorporated herein by reference, a fast refresh can be performed on snapshots defined by a query that includes a subquery. For example, a subquery snapshot on table T1 **1000** may be defined by the following snapshot definition query:

Query 4

select * from T1 where exists (select a from T2 where T1.a=T2.a)

The nested select statement "(select a from T2 where T1.a=T2.a)" is a subquery, where table T1 **1000** is a master table and table T2 is another base table. This snapshot contains all the rows of table T1 **1000** in which the value of column a is also found in column a of table T2. If a row is deleted in table T2, then rows in the snapshot that depend on that row are deleted from the snapshot during a fast refresh. For this purpose, it is advantageous to have an index built on column a to efficiently drive the delete operation. A snapshot definition query can be more complex; for example, the following snapshot definition query has five different subqueries:

Query 5

select * from T1
- where exists (select a from T2 where T1.a=T2.a)
- and exists (select c from T3 where T1.c=T3.c)
- and exists (select b, c from T4 where T1.b=T4.b and T1.c=T4.c)
- and exists (select a, b, c from T5 where T1.a=T5.a and T1.b=T5.b and T1.c=T5.c)
- and exists (select a, b, c, d from T6 where T1.a=T6.a and T1.b=T6.b and T1.c=T6.c and T1.d=T6.d)

For QUERY 5, five column combinations are anticipated to be frequently referenced, namely column combinations {a}, {c}, {b, c}, {a, b, c}, and {a, b, c, d}. Although building five indexes for the respective column combinations enables efficient operation of the fast refresh, the number of indexes that are built is excessive, because the five indexes have to be updated each time a row is deleted from the snapshot. Accordingly, it is desirable to share multicolumn indexes for the column combinations if possible, thereby avoiding unnecessary index maintenance costs.

Use of the present invention to determine the minimum number of indexes to built that can cover the anticipated column combinations in the subqueries of a subquery snapshot advantageously reduces index maintenance costs by eliminating the unnecessary indexes. As described herein above with respect to the working example, one minimal set of indexes for the family of column combinations {a}, {c}, {b, c}, {a, b, c}, and {a, b, c, d} includes an index built on column a and a multi-column index built upon columns c, b, a, and d. Consequently, only two indexes need be maintained, not five indexes according to one conventional approach nor even three indexes according to a use of a depth-first search to find an initial spanning tree on an equivalent directed, acyclic graph.

In the preceding description, the term "column" has been used to refer to columns of relational database tables. However, the term more generally applies to fields into which records from a body of data are organized. For example, in object oriented environments, attributes of object classes act as columns in that they divide object data from objects that belong to the classes into fields. Thus, the present invention is not limited to use with relational tables.

In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will be apparent, however, that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

What is claimed is:

**1**. A method of creating one or more indexes for a body of data arranged in columns, said indexes used to support a plurality of query types, said query types referencing respective combinations of one or more of said columns, said method comprising the computer-implemented steps of:

building a graph based on the plurality of combinations of one or more of said columns of said body of data;

finding a minimum leaf spanning tree for the graph; and

creating said one or more indexes based on the minimum leaf spanning tree.

**2**. The method of claim 1, wherein:

the step of building a graph includes the step of building the graph having a plurality of nodes corresponding respectively to the plurality of combinations of one or more of said columns; and

the step of creating said one or more indexes includes the steps of:

selecting one or more combinations of one or more columns, said combinations of one or more columns corresponding to leaf nodes of the minimum leaf spanning tree; and

creating said one or more indexes based on the one or more selected combinations of one or more columns, respectively.

**3**. The method of claim **2**, wherein the step of building the graph having a plurality of nodes corresponding respectively to the plurality of combinations of one or more of said columns includes the step of adding an edge directed from a first node to a second node, wherein:

the first node corresponds to a first combination of one or more of said columns;

the second node corresponds to a second combination of columns; and

the first combination of columns is a subset of the second combination of columns.

**4**. The method of claim **2**, wherein the step of finding a minimum leaf spanning tree for the graph includes the computer-implemented steps of:

(a) finding an initial spanning tree for the graph;

(b) establishing the initial spanning tree as a current spanning tree;

(c) determining whether an augmenting path exists for the graph and the current spanning tree;

(d) if the augmenting path exists, then determining a new spanning tree, having fewer leaves than the current spanning tree, based on the augmenting path, the current spanning tree, and the graph and establishing the new spanning tree as the current spanning tree;

(e) repeating steps (c) and (d) until no augmenting path exists for the graph and the current spanning tree; and

(f) establishing the current spanning tree as the minimum leaf spanning tree.

**5**. The method of claim **4**, wherein the step of finding an initial spanning tree for the graph includes the step of finding the initial spanning tree by a depth-first search.

**6**. The method of claim **4**, wherein the step of determining whether an augmenting path exists for the graph and the current spanning tree includes the steps of:

(1) establishing a leaf node in the current spanning tree as a current leaf node;

(2) determining whether an augmenting path starting from the current leaf node exists for the graph and the current spanning tree; and

(3) repeating steps (1) and (2) until (3a) the augmenting path starting from the current leaf node exists or (3b) all leaf nodes in the current spanning tree have been considered.

**7**. The method of claim **6**, wherein the step of determining whether an augmenting path starting from the current leaf node exists for the graph and the current spanning tree includes the steps of:

(i) establishing the current leaf node as a current node;

(ii) determining whether there exist a first edge in the graph but not in the current spanning tree directed from the current node to a first node and a second edge in the current spanning tree directed from a second node to the first node, wherein the second node is not already part of the augmenting path starting from the current leaf node;

(iii) if neither said first edge nor said second edge exists for the current node, then establishing that the augmenting path starting from the current leaf node does not exist;

(iv) if both said first edge and said second edge exist for the current node, then determining whether there exists a third edge in the current spanning tree directed from the second node to a third node different from the first node;

(v) if both said first edge and said second edge exist for the current node and the third edge exists, then establishing that the augmenting path starting from the current leaf node including the first node, the second node, and the third node exists;

(vi) if both said first edge and said second edge exist for the current node and the third edge does not exist, then determining whether a continuation of the augmenting path starting from the second node exists for the graph and the current spanning tree; and

(vii) if the continuation of the augmenting path exists, then establishing that the augmenting path starting from the current leaf node including the first node, the second node, and the continuation of the augmenting path exists.

**8**. The method of claim **2**, wherein the step of building the graph having a plurality of nodes corresponding respectively to the plurality of combinations of one or more of said columns includes the step of building the graph with a root node corresponding to an empty combination and a plurality of edges directed from the root node to the plurality of nodes, respectively.

**9**. The method of claim **1**, wherein the step of building a graph based on the plurality of the combinations of one or more of said columns of said body of data includes the step of building the graph based on a snapshot definition query.

**10**. A method of finding a minimum leaf spanning tree for a directed acyclic graph (DAG), said method comprising the computer-implemented steps of:

(a) finding an initial spanning tree for the DAG;

(b) establishing the initial spanning tree as a current spanning tree;

(c) determining whether an augmenting path exists for the DAG and the current spanning tree;

(d) if the augmenting path exists, then determining a new spanning tree, having fewer leaves than the current spanning tree, based on the augmenting path, the current spanning tree, and the graph and establishing the new spanning tree as the current spanning tree;

(e) repeating steps (c) and (d) until no augmenting path exists for the DAG and the current spanning tree; and

(f) establishing the current spanning tree as the minimum leaf spanning tree.

**11**. The method of claim **10**, wherein the step of finding an initial spanning tree for the DAG includes the step of finding the initial spanning tree by a depth-first search.

**12**. The method of claim **10**, wherein the step of determining whether an augmenting path exists for the DAG and the current spanning tree includes the steps of:

(1) establishing a leaf node in the current spanning tree as a current leaf node;

(2) determining whether an augmenting path starting from the current leaf node exists for the DAG and the current spanning tree; and

(3) repeating steps (1) and (2) until (3a) the augmenting path starting from the current leaf node exists or (3b) all leaf nodes in the current spanning tree have been considered.

**13**. The method of claim **12**, wherein the step of determining whether an augmenting path starting from the current leaf node exists for the DAG and the current spanning tree includes the steps of:

(i) establishing the current leaf node as a current node;

(ii) determining whether there exist a first edge in the DAG but not in the current spanning tree directed from the current node to a first node and a second edge in the current spanning tree directed from a second node to the first node, wherein the second node is not already part of the augmenting path starting from the current leaf node;

(iii) if neither said first edge nor said second edge exists for the current node, then establishing that the augmenting path starting from the current leaf node does not exist;

(iv) if both said first edge and said second edge exist for the current node, then determining whether there exists a third edge in the current spanning tree directed from the second node to a third node different from the first node;

(v) if both said first edge and said second edge exist for the current node and the third edge exists, then establishing that the augmenting path starting from the current leaf node including the first node, the second node, and the third node exists;

(vi) if both said first edge and said second edge exist for the current node and the third edge does not exist, then determining whether a continuation of the augmenting path starting from the second node exists for the DAG and the current spanning tree; and

(vii) if the continuation of the augmenting path exists, then establishing that the augmenting path starting from the current leaf node including the first node, the second node, and the continuation of the augmenting path exists.

**14**. A computer-readable medium bearing instructions for creating one or more indexes for a body of data arranged in columns, said indexes used to support a plurality of query types, said query types referencing respective combinations of one or more of said columns, said instructions arranged to cause one or more processors to perform the steps of:

building a graph based on the plurality of combinations of one or more of said columns of said body of data;

finding a minimum leaf spanning tree for the graph; and

creating said one or more indexes based on the minimum leaf spanning tree.

**15**. The computer-readable medium of claim **14**, wherein:

the step of building a graph includes the step of building the graph having a plurality of nodes corresponding respectively to the plurality of combinations of one or more of said columns; and

the step of creating said one or more indexes includes the steps of:

selecting one or more combinations of one or more columns, said combinations of one or more columns corresponding to leaf nodes of the minimum leaf spanning tree; and

creating said one or more indexes based on the one or more selected combinations of one or more columns, respectively.

**16**. The computer-readable medium of claim **15**, wherein the step of building the graph having a plurality of nodes corresponding respectively to the plurality of combinations of one or more of said columns includes the step of adding an edge directed from a first node to a second node, wherein:

the first node corresponds to a first combination of one or more of said columns;

the second node corresponds to a second combination of columns; and

the first combination of columns is a subset of the second combination of columns.

**17**. The computer-readable medium of claim **15**, wherein the step of finding a minimum leaf spanning tree for the graph includes the computer-implemented steps of:

(a) finding an initial spanning tree for the graph;

(b) establishing the initial spanning tree as a current spanning tree;

(c) determining whether an augmenting path exists for the graph and the current spanning tree;

(d) if the augmenting path exists, then determining a new spanning tree, having fewer leaves than the current spanning tree, based on the augmenting path, the current spanning tree, and the graph and establishing the new spanning tree as the current spanning tree;

(e) repeating steps (c) and (d) until no augmenting path exists for the graph and the current spanning tree; and

(f) establishing the current spanning tree as the minimum leaf spanning tree.

**18**. The computer-readable medium of claim **17**, wherein the step of finding an initial spanning tree for the graph includes the step of finding the initial spanning tree by a depth-first search.

**19**. The computer-readable medium of claim **17**, wherein the step of determining whether an augmenting path exists for the graph and the current spanning tree includes the steps of:

(1) establishing a leaf node in the current spanning tree as a current leaf node;

(2) determining whether an augmenting path starting from the current leaf node exists for the graph and the current spanning tree; and

(3) repeating steps (1) and (2) until (3a) the augmenting path starting from the current leaf node exists or (3b) all leaf nodes in the current spanning tree have been considered.

**20**. The computer-readable medium of claim **19**, wherein the step of determining whether an augmenting path starting from the current leaf node exists for the graph and the current spanning tree includes the steps of:

(i) establishing the current leaf node as a current node;

(ii) determining whether there exist a first edge in the graph but not in the current spanning tree directed from the current node to a first node and a second edge in the current spanning tree directed from a second node to the first node, wherein the second node is not already part of the augmenting path starting from the current leaf node;

(iii) if neither said first edge nor said second edge exists for the current node, then establishing that the augmenting path starting from the current leaf node does not exist;

(iv) if both said first edge and said second edge exist for the current node, then determining whether there exists a third edge in the current spanning tree directed from the second node to a third node different from the first node;

(v) if both said first edge and said second edge exist for the current node and the third edge exists, then establishing that the augmenting path starting from the current leaf node including the first node, the second node, and the third node exists;

(vi) if both said first edge and said second edge exist for the current node and the third edge does not exist, then determining whether a continuation of the augmenting path starting from the second node exists for the graph and the current spanning tree; and

(vii) if the continuation of the augmenting path exists, then establishing that the augmenting path starting from the current leaf node including the first node, the second node, and the continuation of the augmenting path exists.

**21**. The computer-readable medium of claim **15**, wherein the step of building the graph having a plurality of nodes corresponding respectively to the plurality of combinations of one or more of said columns includes the step of building the graph with a root node corresponding to an empty combination and a plurality of edges directed from the root node to the plurality of nodes, respectively.

**22**. The computer-readable medium of claim **14**, wherein the step of building a graph based on the plurality of the combinations of one or more of said columns of said body of data includes the step of building the graph based on a snapshot definition query.

**23**. A computer-readable medium bearing instructions for finding a minimum leaf spanning tree for a directed acyclic graph (DAG), said instructions arranged to cause one or more processors to perform the steps of:

(a) finding an initial spanning tree for the DAG;

(b) establishing the initial spanning tree as a current spanning tree;

(c) determining whether an augmenting path exists for the DAG and the current spanning tree;

(d) if the augmenting path exists, then determining a new spanning tree, having fewer leaves than the current spanning tree, based on the augmenting path, the current spanning tree, and the graph and establishing the new spanning tree as the current spanning tree;

(e) repeating steps (c) and (d) until no augmenting path exists for the DAG and the current spanning tree; and

(f) establishing the current spanning tree as the minimum leaf spanning tree.

**24**. The computer-readable medium of claim **23**, wherein the step of finding an initial spanning tree for the DAG

includes the step of finding the initial spanning tree by a depth-first search.

**25**. The computer-readable medium of claim **23**, wherein the step of determining whether an augmenting path exists for the DAG and the current spanning tree includes the steps of:

(1) establishing a leaf node in the current spanning tree as a current leaf node;

(2) determining whether an augmenting path starting from the current leaf node exists for the DAG and the current spanning tree; and

(3) repeating steps (1) and (2) until (3a) the augmenting path starting from the current leaf node exists or (3b) all leaf nodes in the current spanning tree have been considered.

**26**. The computer-readable medium of claim **25**, wherein the step of determining whether an augmenting path starting from the current leaf node exists for the DAG and the current spanning tree includes the steps of:

(i) establishing the current leaf node as a current node;

(ii) determining whether there exist a first edge in the DAG but not in the current spanning tree directed from the current node to a first node and a second edge in the current spanning tree directed from a second node to the first node, wherein the second node is not already part of the augmenting path starting from the current leaf node;

(iii) if neither said first edge nor said second edge exists for the current node, then establishing that the augmenting path starting from the current leaf node does not exist;

(iv) if both said first edge and said second edge exist for the current node, then determining whether there exists a third edge in the current spanning tree directed from the second node to a third node different from the first node;

(v) if both said first edge and said second edge exist for the current node and the third edge exists, then establishing that the augmenting path starting from the current leaf node including the first node, the second node, and the third node exists;

(vi) if both said first edge and said second edge exist for the current node and the third edge does not exist, then determining whether a continuation of the augmenting path starting from the second node exists for the DAG and the current spanning tree; and

(vii) if the continuation of the augmenting path exists, then establishing that the augmenting path starting from the current leaf node including the first node, the second node, and the continuation of the augmenting path exists.

* * * * *

US006138123A

# United States Patent [19]

Rathbun

[11] Patent Number: 6,138,123

[45] Date of Patent: *Oct. 24, 2000

[54] METHOD FOR CREATING AND USING PARALLEL DATA STRUCTURES

[76] Inventor: Kyle R. Rathbun, 2357 Stonehedge Dr., Apt. E, East Lansing, Mich. 48823

[ * ] Notice: This patent issued on a continued prosecution application filed under 37 CFR 1.53(d), and is subject to the twenty year patent term provisions of 35 U.S.C. 154(a)(2).

[21] Appl. No.: 08/892,705

[22] Filed: Jul. 15, 1997

102(e).

Related U.S. Application Data

[60] Provisional application No. 60/023,340, Jul. 25, 1996, and provisional application No. 60/022,616, Jul. 26, 1996.

[51] Int. Cl.[7] ..... G06F 17/30

[52] U.S. Cl. ..... 707/201; 707/102; 345/339; 345/800

[58] Field of Search ..... 707/102, 201, 707/5, 104, 2, 3, 4, 101, 531, 7, 8, 10; 395/800, 200; 364/231, 490, 468; 455/456; 370/381, 389, 256, 406; 711/129, 153, 173, 206; 345/339, 349, 440, 800; 358/1, 18

[56] References Cited

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 5,230,047 | 7/1993 | Frey, Jr. | 395/182 |
| 5,319,778 | 6/1994 | Catino | 707/102 |
| 5,430,869 | 7/1995 | Ishak et al. | 707/101 |
| 5,475,837 | 12/1995 | Ishak et al. | 707/101 |
| 5,475,851 | 12/1995 | Kodosky et al. | 345/339 |
| 5,535,408 | 7/1996 | Hillis | 345/800 |
| 5,539,922 | 7/1996 | Wang | 455/456 |
| 5,551,027 | 8/1996 | Choy et al. | 707/201 |
| 5,602,754 | 2/1997 | Beatty et al. | 364/489 |
| 5,608,903 | 3/1997 | Prasad et al. | 707/10 |

OTHER PUBLICATIONS

R.G. Gallager et al., "A Distributed Algorithm for Minimum–Weight Spanning Trees", Jan. 1983, *ACM Transactions on Programming Languages and Systems*, vol. 5, No. 1, pp. 66–77.

Richard Weinberg, "Parallel Processing Image Synthesis and Anti–Aliasing", Aug. 1981, *Computer Graphics*, vol. 15, No. 3, pp. 55–62.

Shmuel Zaks, "Optimal Distributed Algorithms for Sorting and Ranking", Apr. 1985, *IEEE Transactions on Computers*, vol. C–34, No. 4, pp. 376–379.

Clyde P. Kruskal, "Searching, Merging, and Sorting in Parallel Computation", Oct. 1983, *IEEE Transactions on Computers*, vol. C–32, No. 10, pp. 942–946.

Carla Schlatter Ellis, "Distributed Data Structures: A Case Study", May 1985, *The 5th International Conference on Distributed Computing Systems*, IEEE Computer Society, Computer Society Press, pp. 201–208.

Ossama I. El–Dessouki et al., "Distributed Search of Game Trees", May 1984, *The 4th International Conference on Distributed Computing Systems*, IEEE Computer Society, Computer Society Press, pp. 183–191.

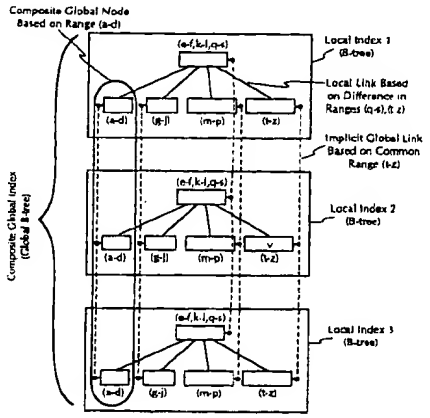
(List continued on next page.)

*Primary Examiner*—Wayne Amsbury
*Assistant Examiner*—Thu-Thao Havan
*Attorney, Agent, or Firm*—Harness, Dickey & Pierce, P.L.C.

[57] ABSTRACT

Parallel data-structures distribute a given data set to system components by grouping the data set according to ranges. These ranges are sub-divided for distribution into parallel form. A given data value is located by its placement within an appropriate range; the ranges are located by their relationships to each other and the data set as a whole; thus, the ranges are related to each other, the order of the data set is maintained and access is gained to the data set by range. Each range may be distributed to multiple nodes; each node may be contained in a separate data-structure; each separate data-structure may be maintained on a separate system component. The result is a method of creating and using parallel data-structures that may take a wide variety of forms and be used to control data distribution and the efficient distribution of system resources.

**36 Claims, 68 Drawing Sheets**



OTHER PUBLICATIONS

Raphael Finkel and Udi Manber, "DIB—A Distributed Implementation of Backtracking", May 1985, *The 5th International Conference on Distributed Computing Systems*, IEEE Computer Society, Computer Society Press, pp. 446–452.

W. Daniel Hillis and Guy L. Steele, Jr., "Data Parallel Algorithms", Dec. 1986, *Communications of the ACM*, vol. 29, No. 12, pp. 1170–1183.

Jishnu Mukerji and Richard B. Kieburtz, "A Distributed File System for a Hierarchical Multicomputer", Oct. 1979, *The 1st International Conference on Distributed Computing Systems*, IEEE Computer Society, Catalog No. 79CH1445–6 C, pp. 448–457.

Keki B. Irani et al., "A Combined Communication Network Design and File Allocation for Distributed Databases", Apr. 1981, *The 2nd International Conference on Distributed Computing Systems*, IEEE Catalog No. 81CH1591–7, Computer Society Press, pp. 197–210.

Bruce Lindsay, "Object Naming and Catalog Management for a Distributed Database Manager", Apr. 1981, *The 2nd International Conference on Distributed Computing Systems*, IEEE Catalog No. 81CH1591–7, Computer Society Press, pp. 31–40.

Ajay K. Gupta et al., "Load Balanced Priority Queues on Distributed Memory Machines", Western Michigan University Research, Fellowship from the Faculty Research and Creative Activities Support Funds, WMU–FRCASF 90–15 and WMU–FRACASF 94–040 and National Science Foundation, Grant No. USE–90–52346.

Elise de Doncker et al., "Two Methods for Load Balanced Distributed Adaptive Integration", Department Computer Science, Western Michigan University, National Science Foundation, Grant No. CCR–9405377.

Elise de Doncker et al., "Use of ParInt for Parallel Computation of Statistics Integrals", Department Computer Science, Western Michigan University, National Science Foundation, Grant Nos. CCR–9405377 and DMS–9211640.

Elise de Doncker et al., "Development of a Parallel and Distributed Integration Package—Part I", Department Computer Science, Western Michigan University, National Science Foundation, Grant No. CCR–9405377.

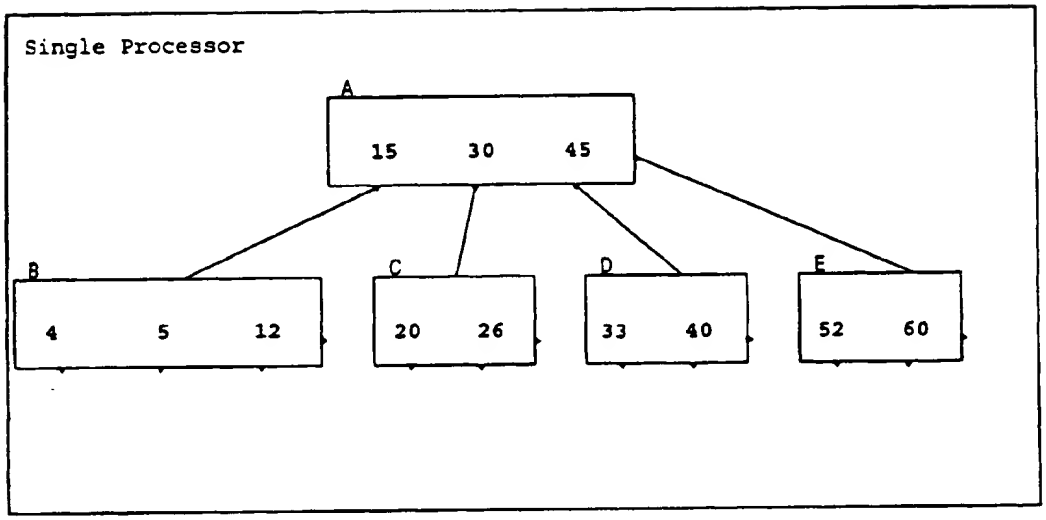



Figure 1

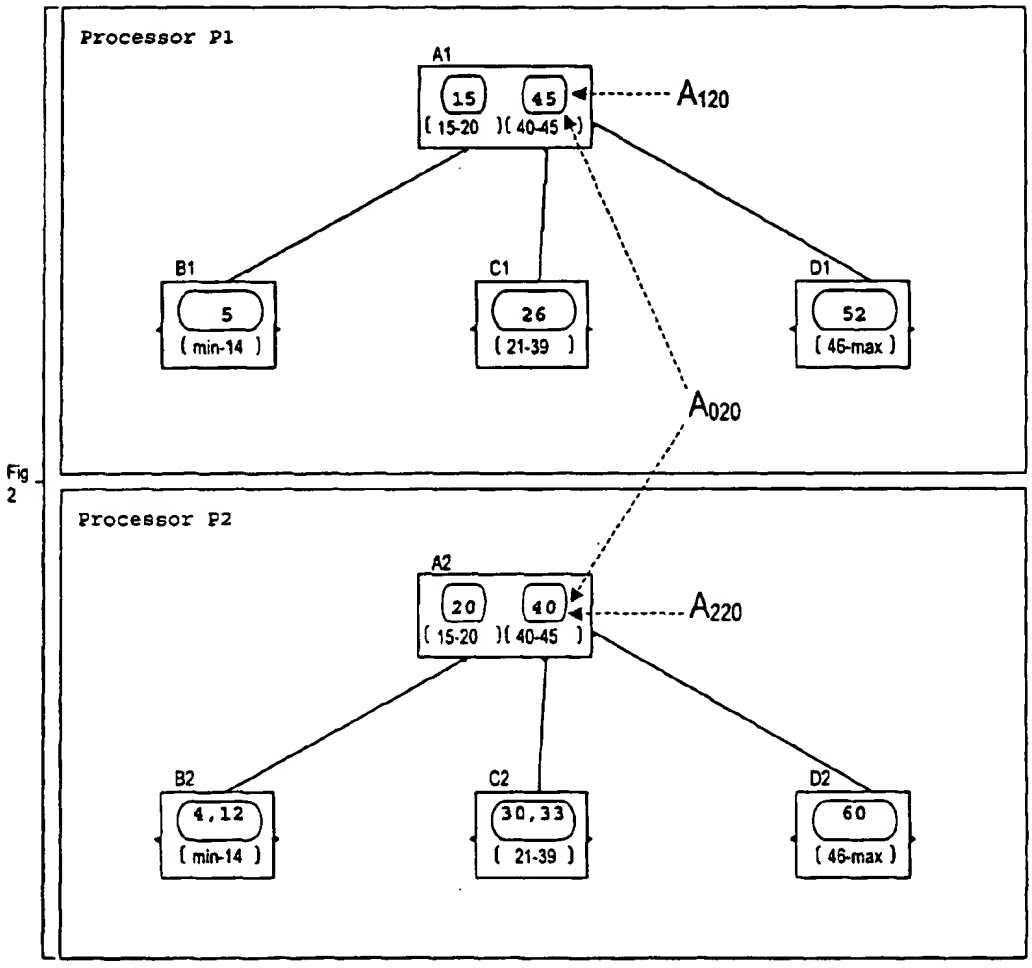



Fig 2

Figure 2

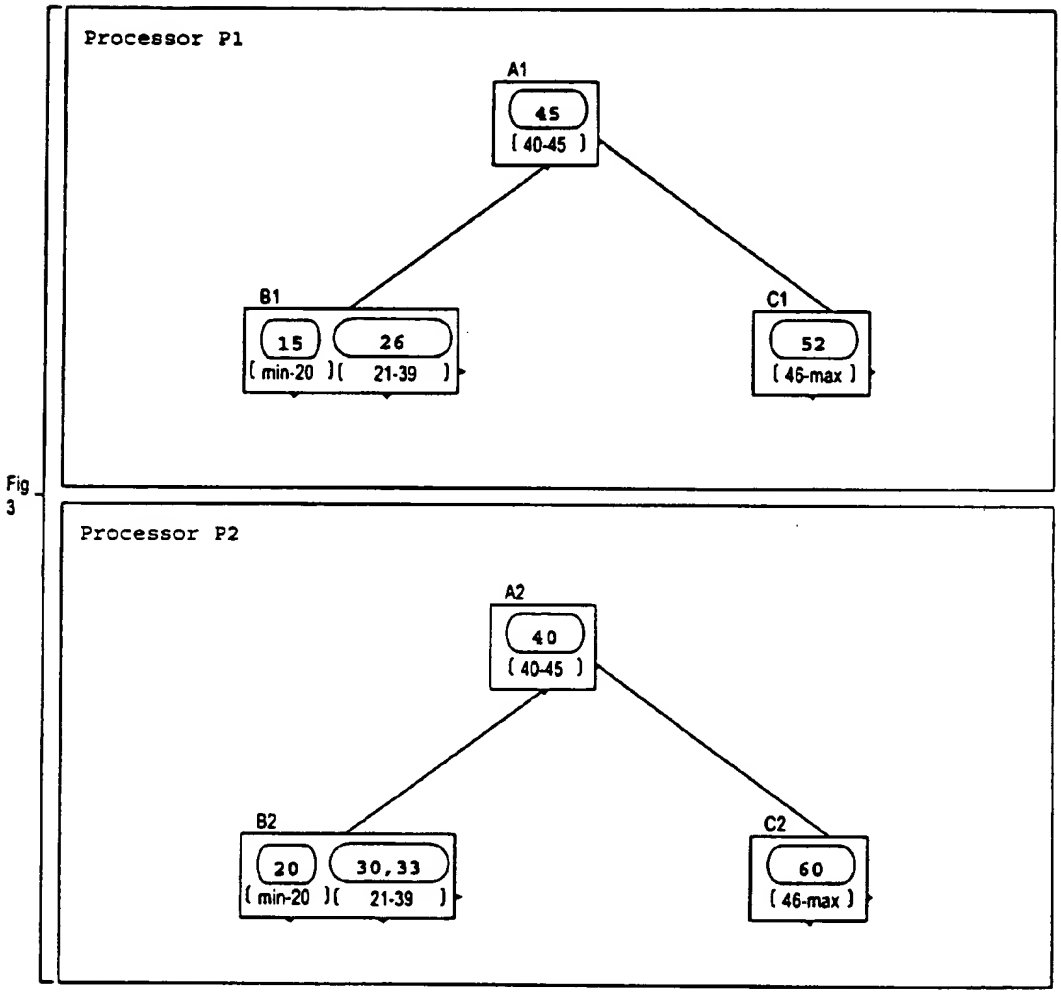



Fig 3

Figure 3

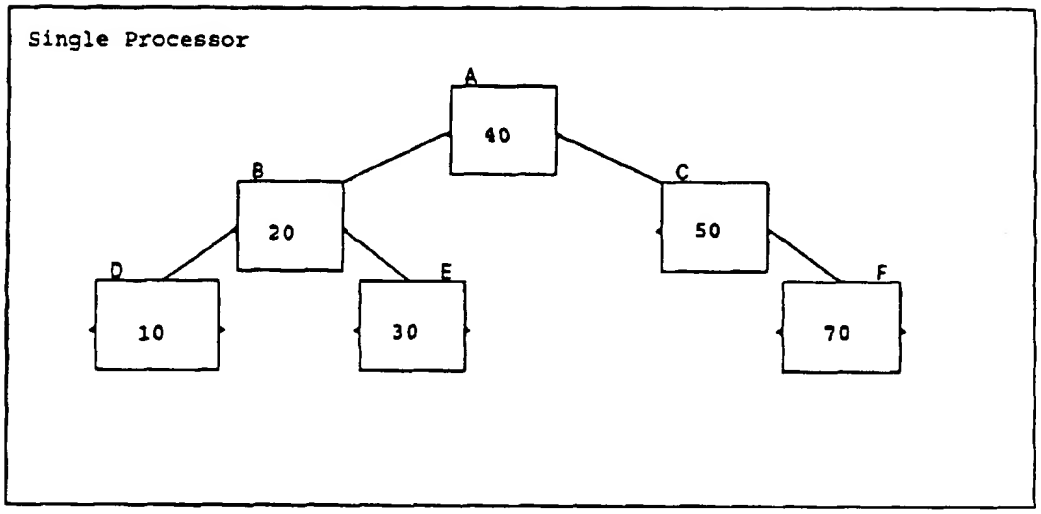



Figure 4

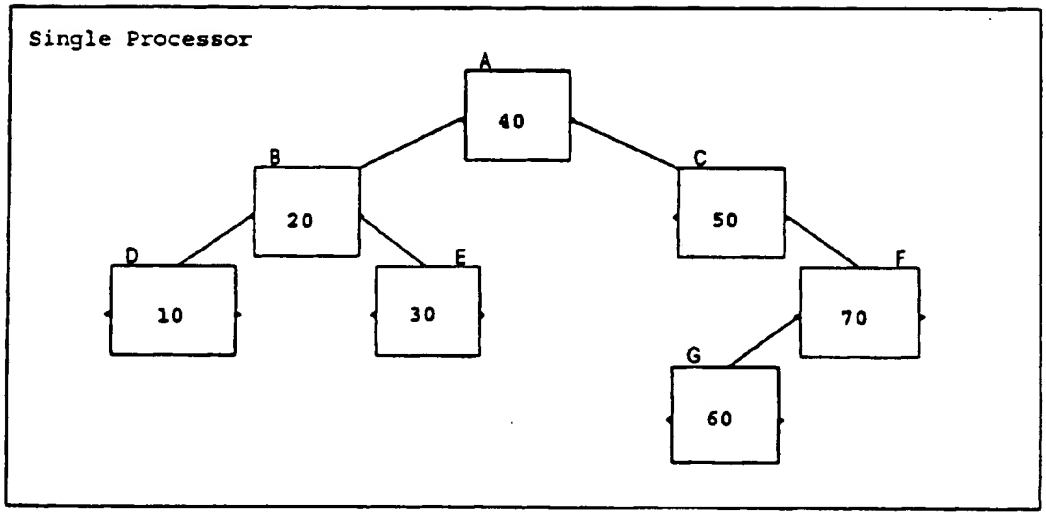



Figure 5

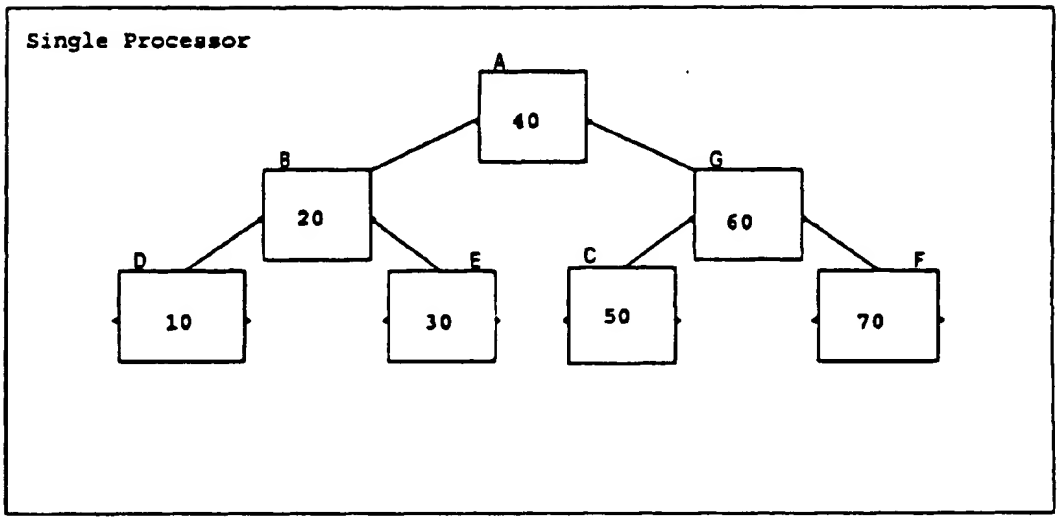



Figure 6

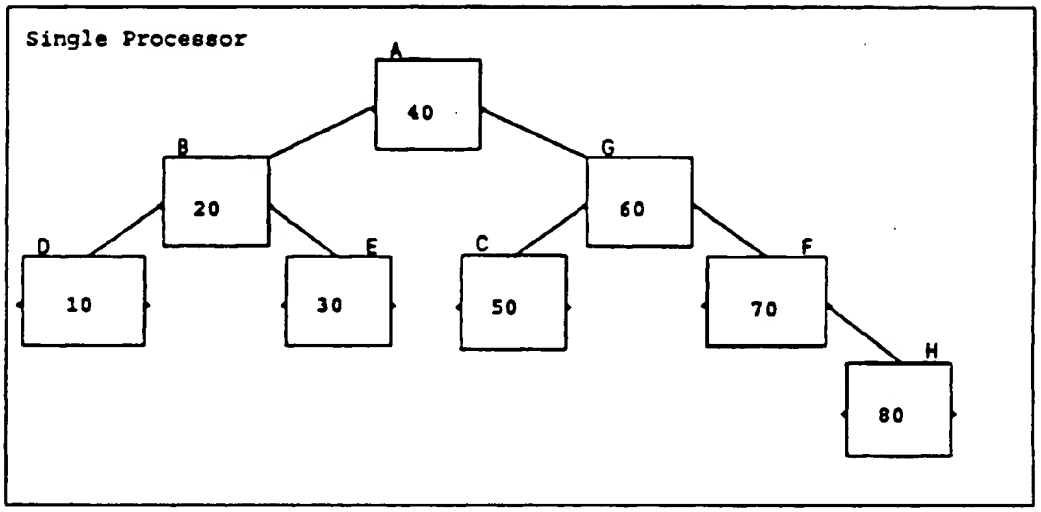



Figure 7

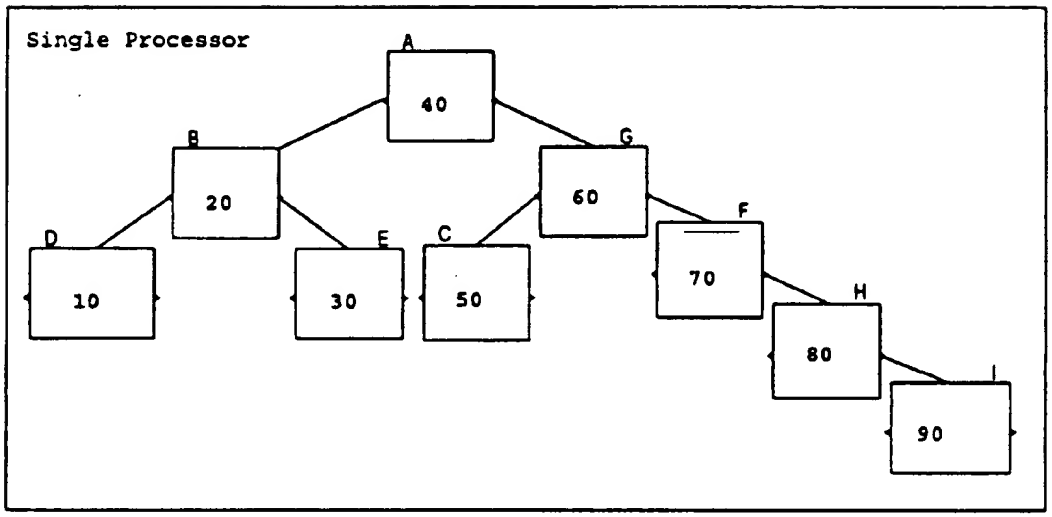



Figure 8

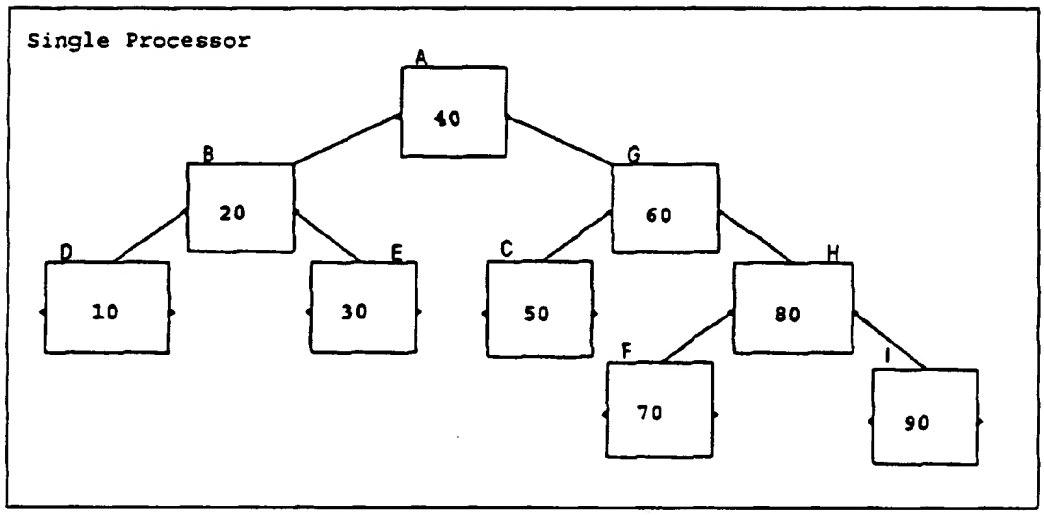



Figure 9

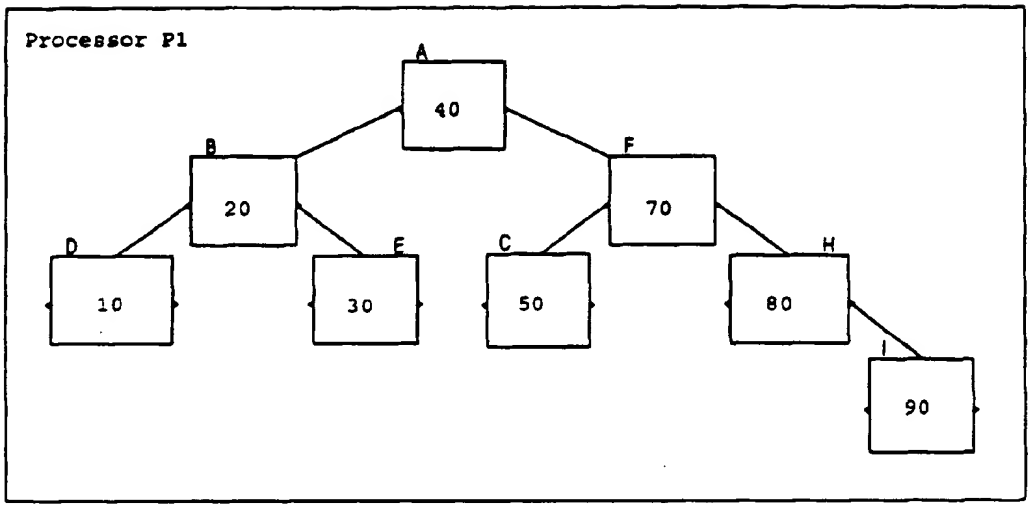



Figure 10

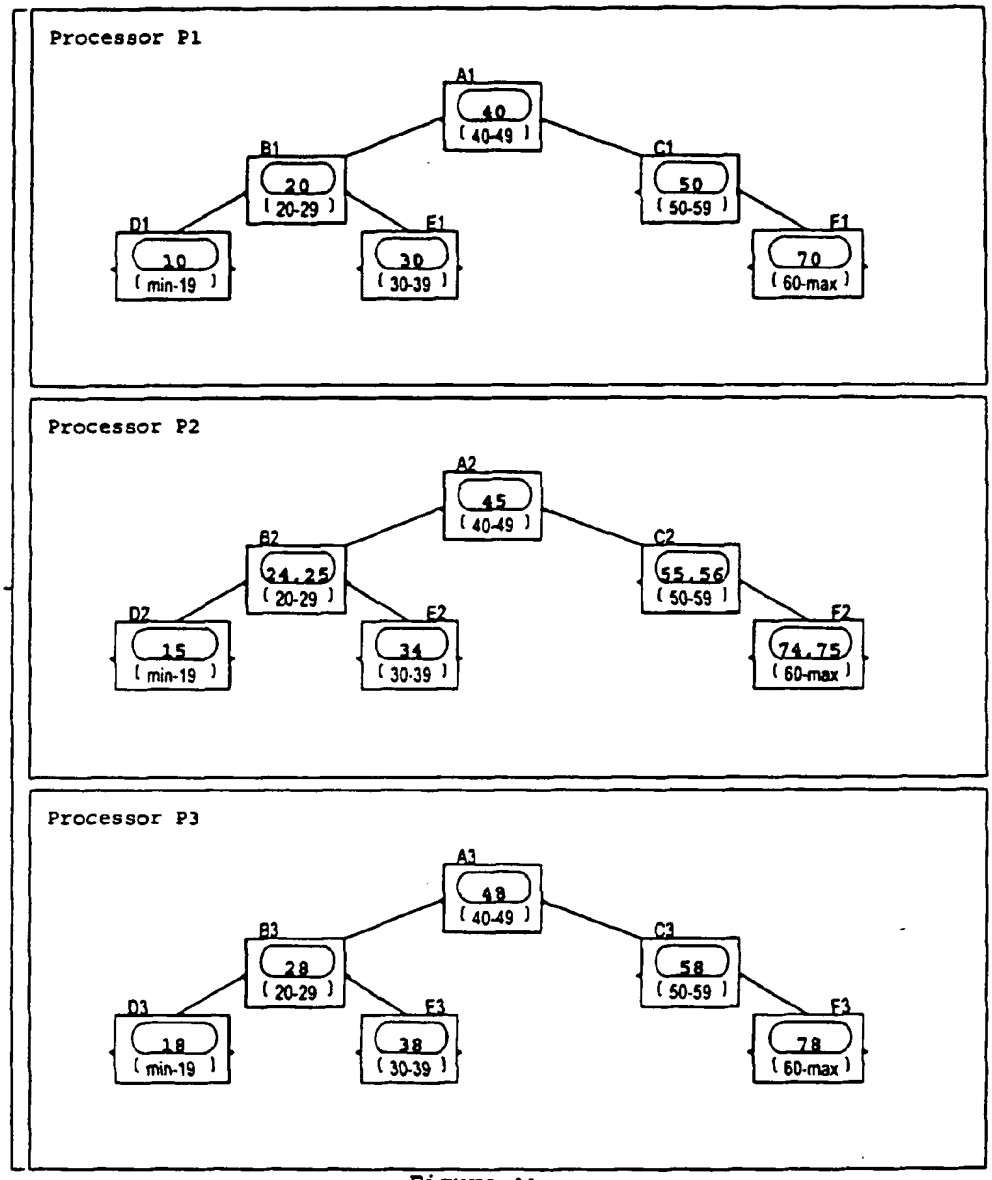



Fig 11

Figure 11

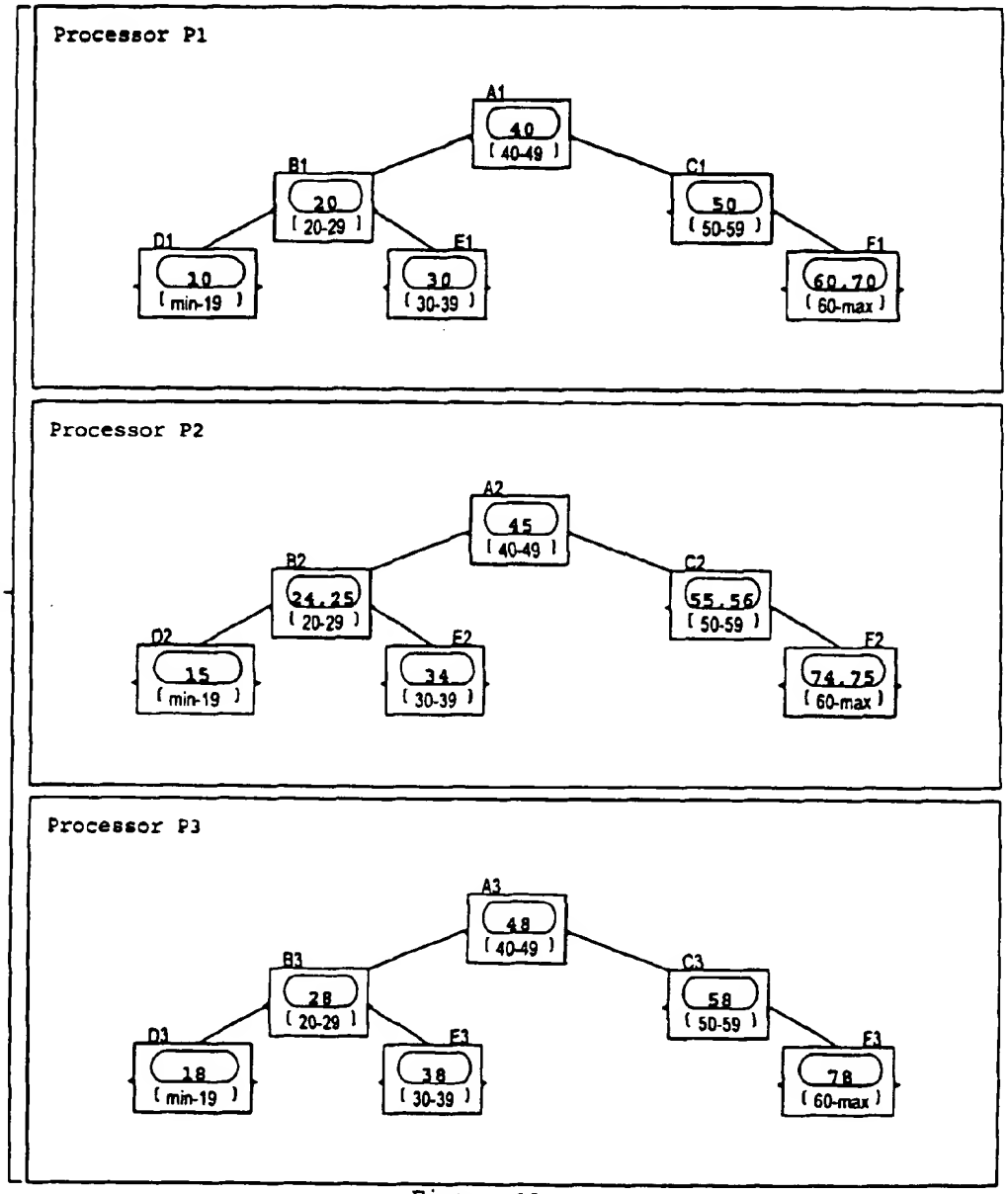



Fig 12

Figure 12

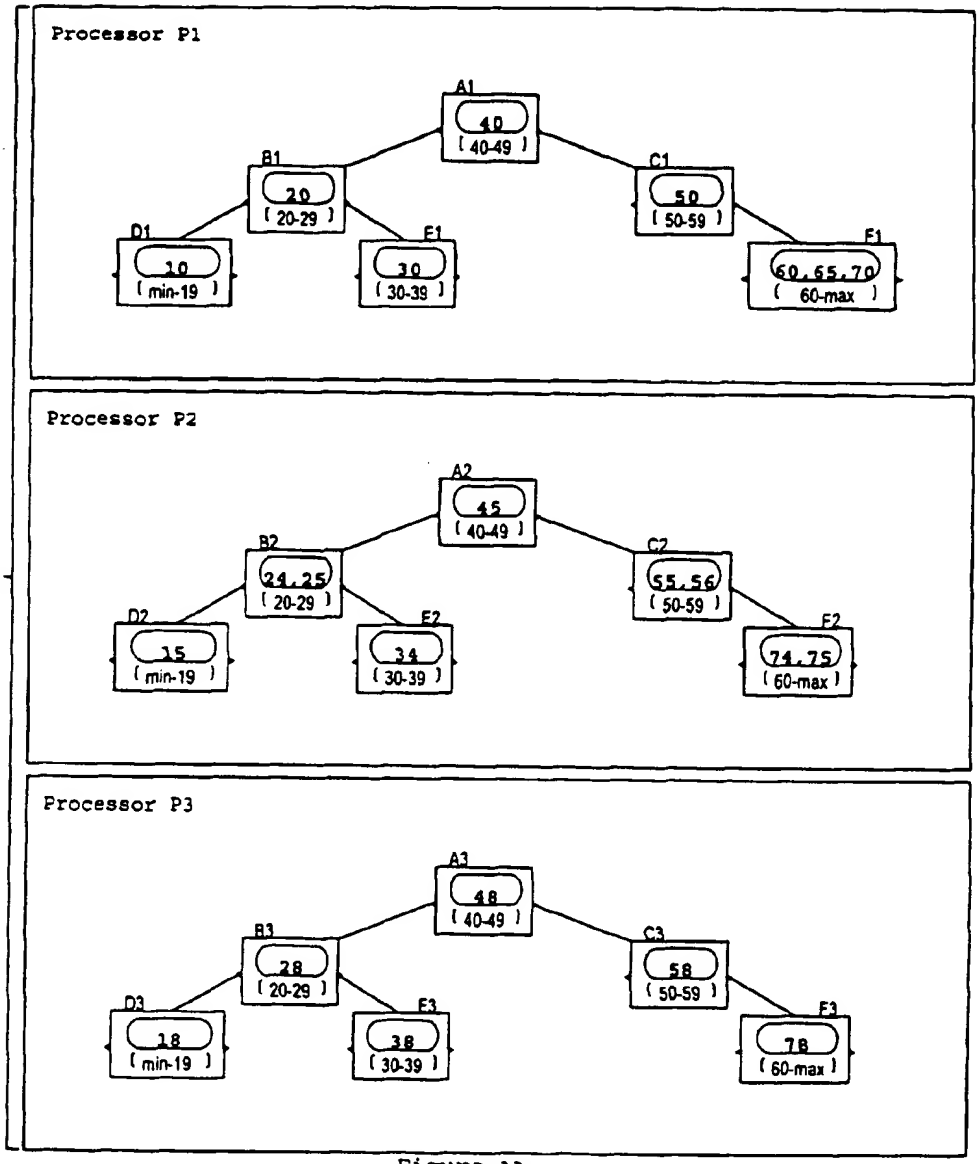



Fig 13

Figure 13

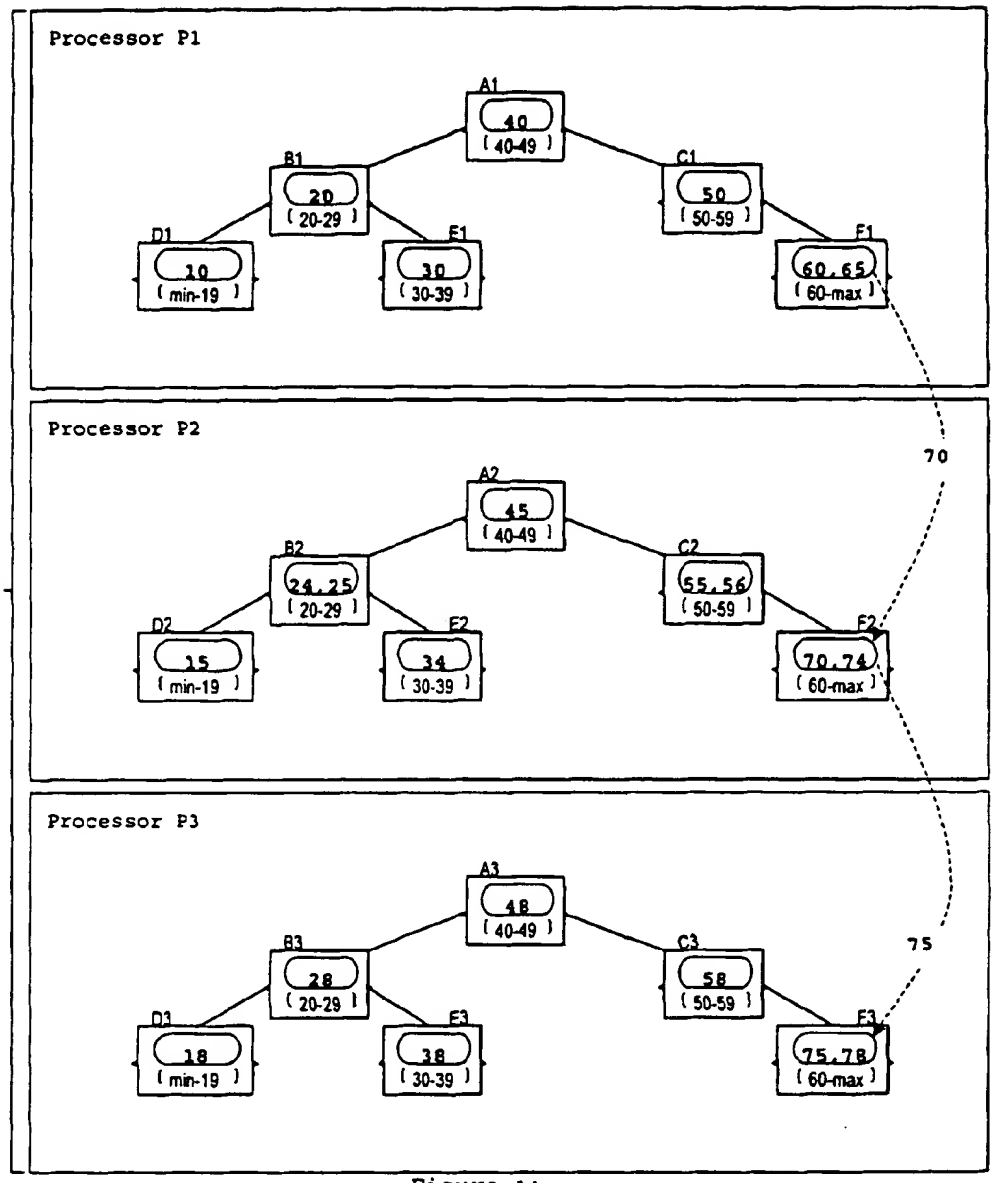
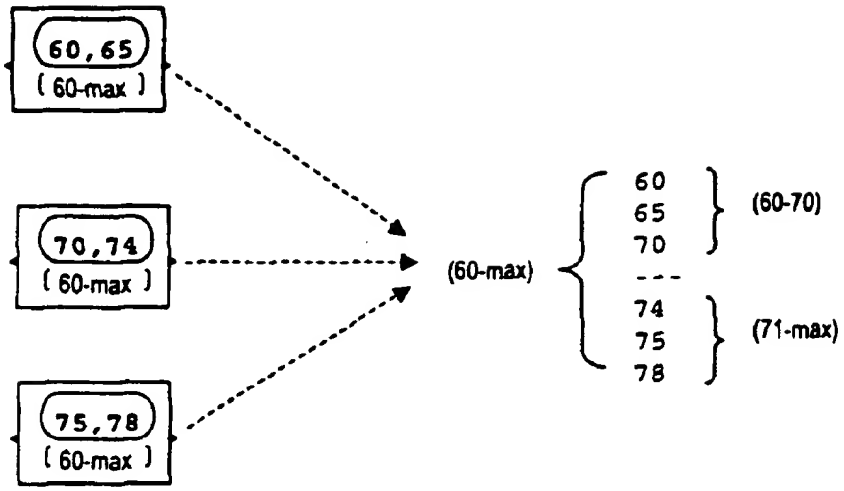



Fig 14

Figure 14


60,65
( 60-max )
70,74
( 60-max )
75,78
( 60-max )
(60-max)
60
65
70
---
74
75
78
(60-70)
(71-max)

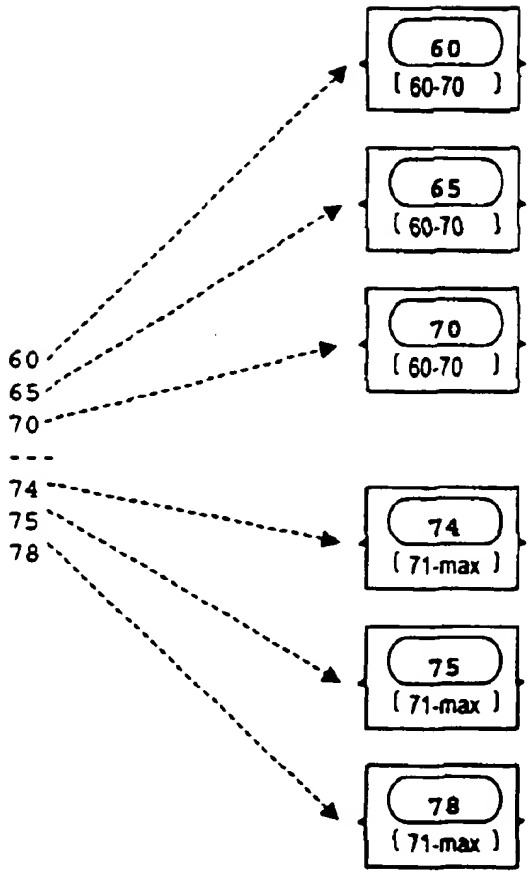

60
65
70
---
74
75
78
60
( 60-70 )
65
( 60-70 )
70
( 60-70 )
74
( 71-max )
75
( 71-max )
78
( 71-max )

Figure 15

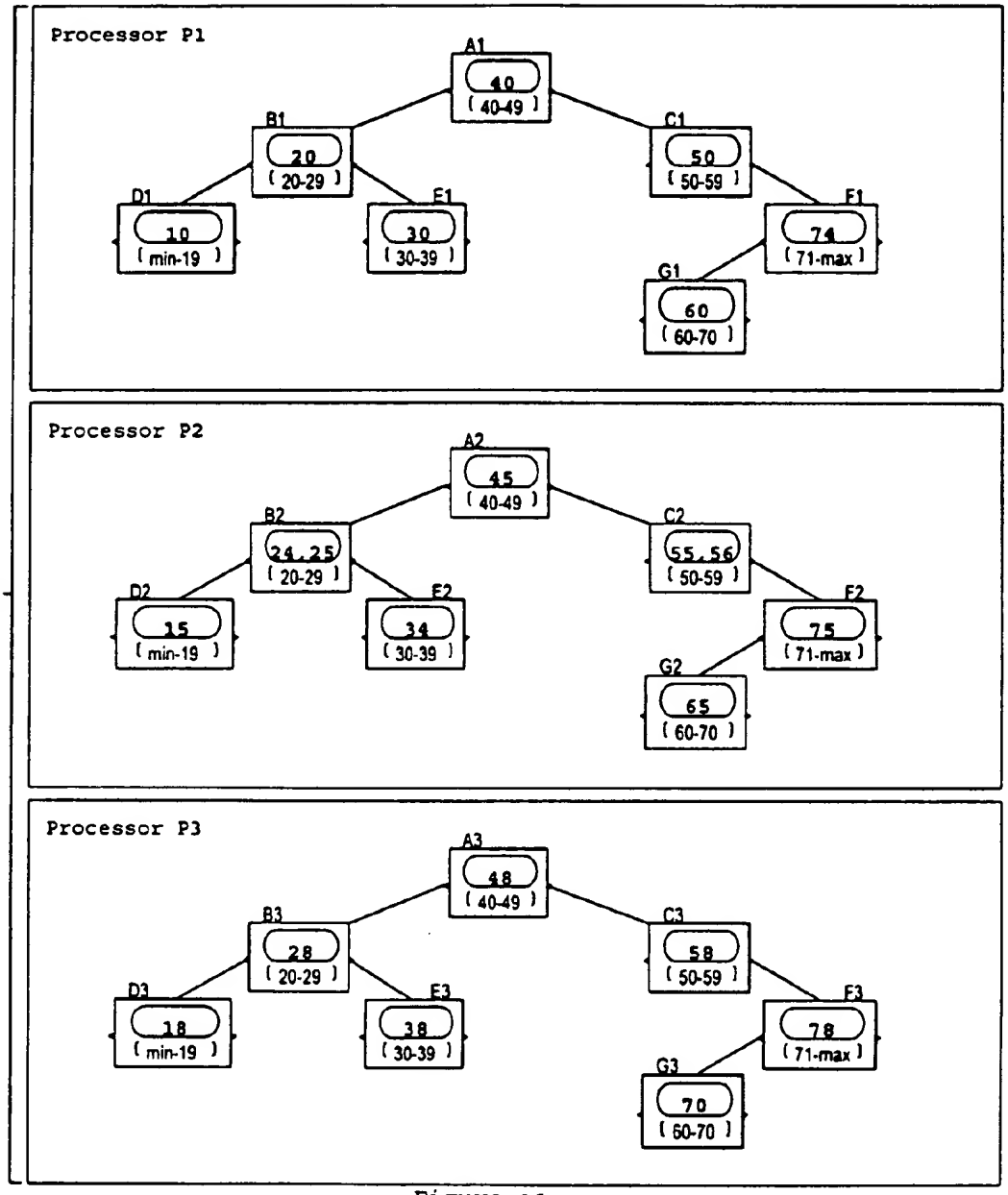



Fig 16

Figure 16

Fig 17

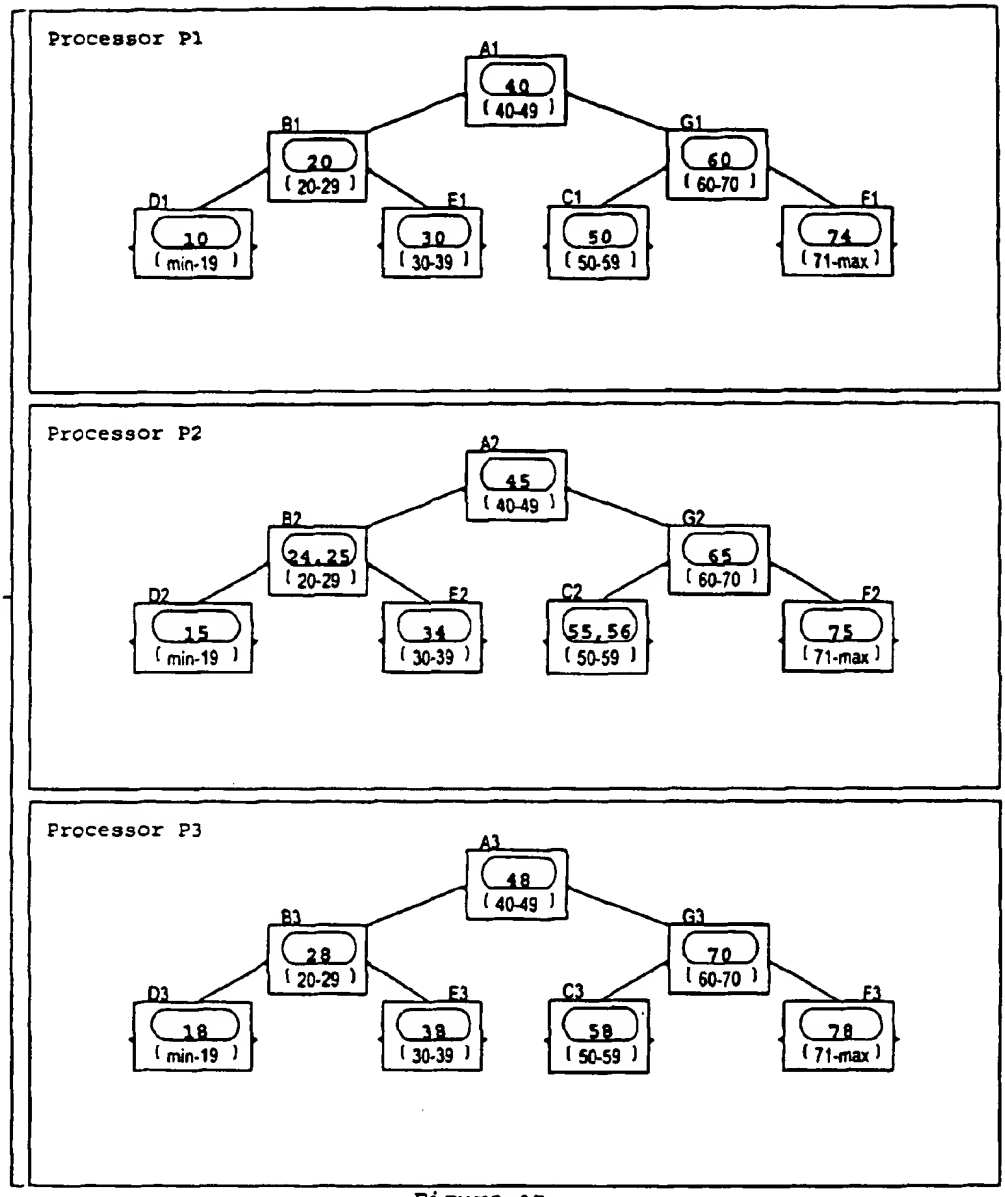



Figure 17

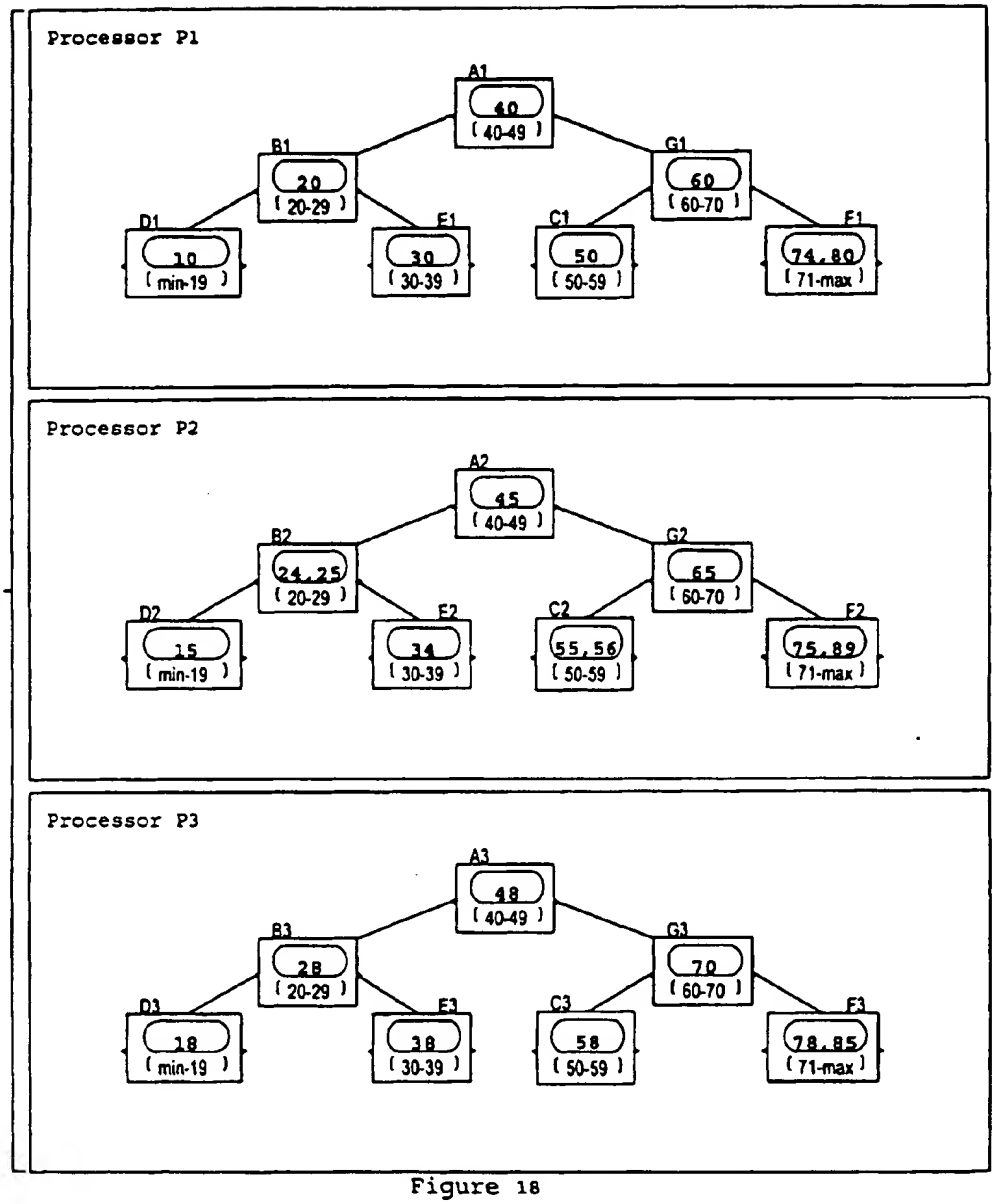



Fig 18

Figure 18

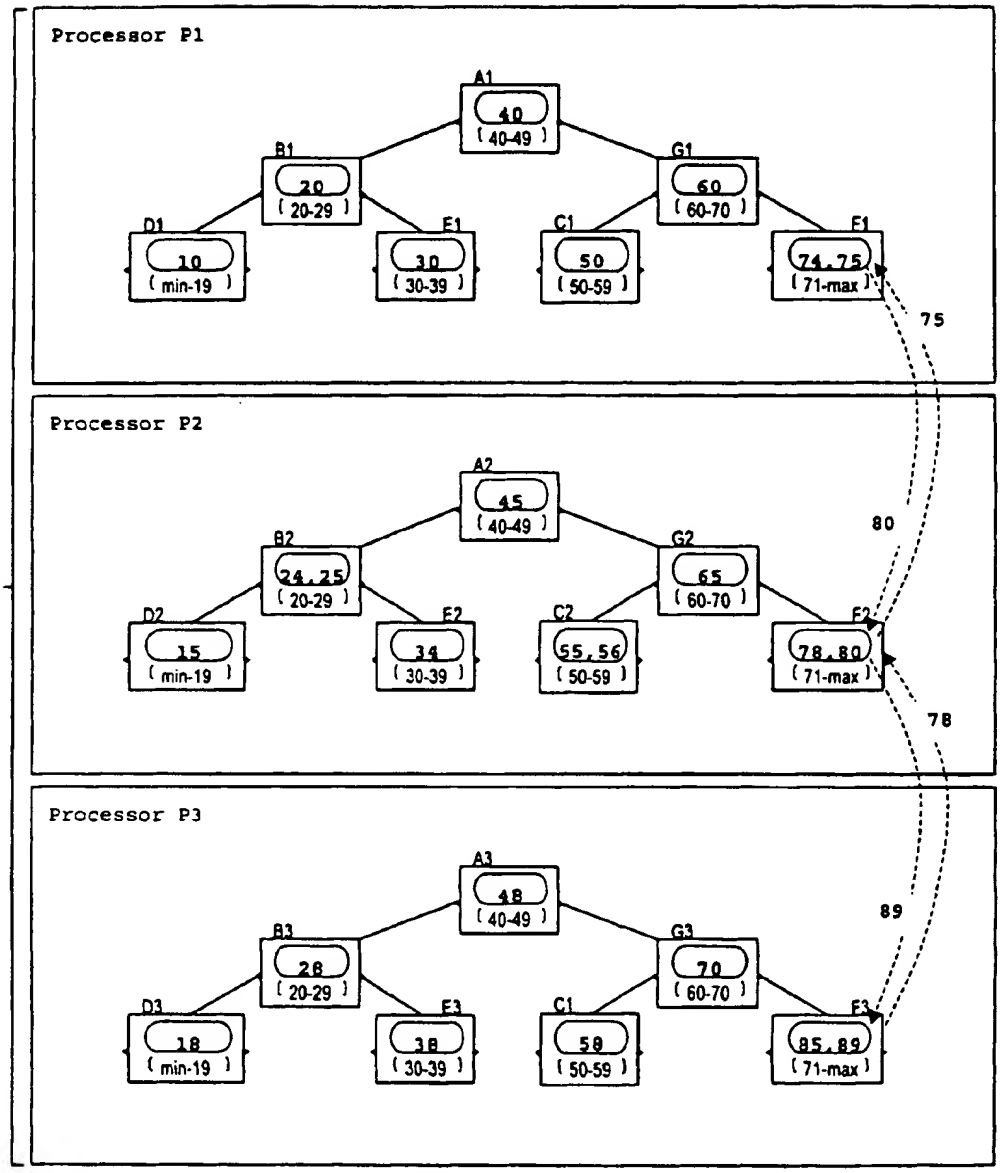
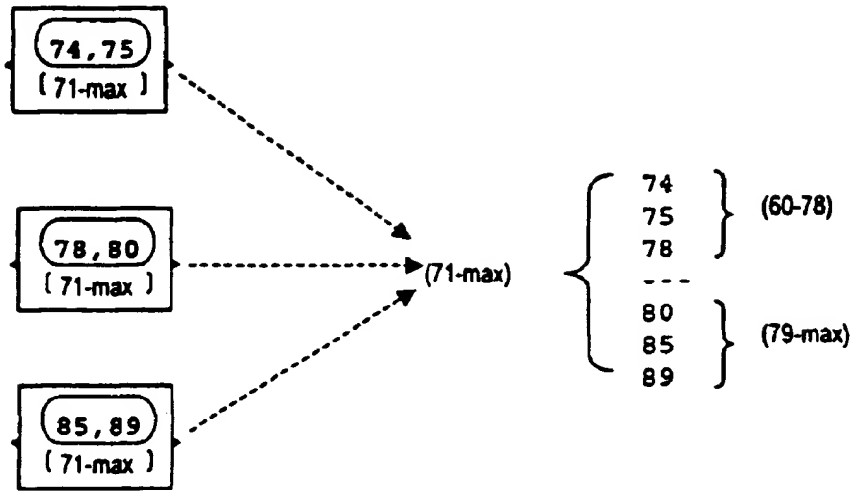



Fig 19

Figure 19


74,75
[71-max]
78,80
[71-max]
85,89
[71-max]
(71-max)
74
75
78
---
80
85
89
(60-78)
(79-max)

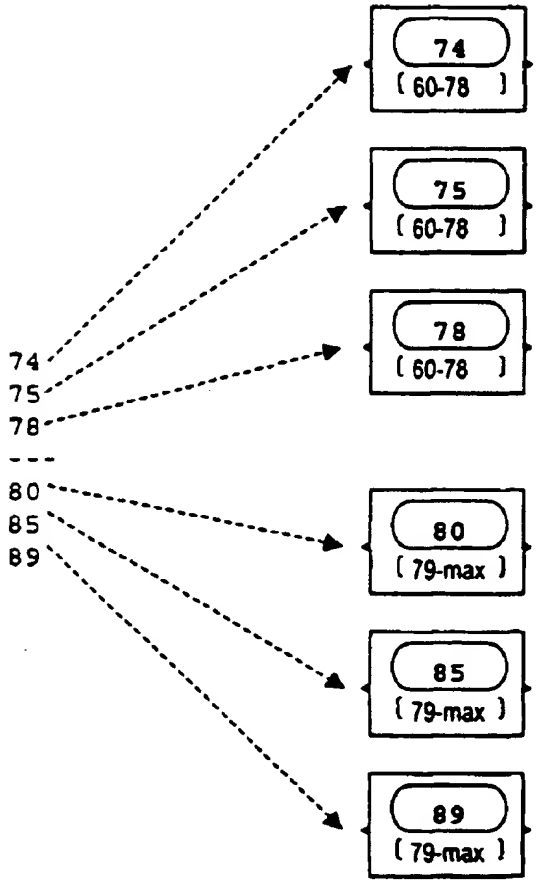

74
75
78
---
80
85
89
74
[60-78]
75
[60-78]
78
[60-78]
80
[79-max]
85
[79-max]
89
[79-max]

Figure 20

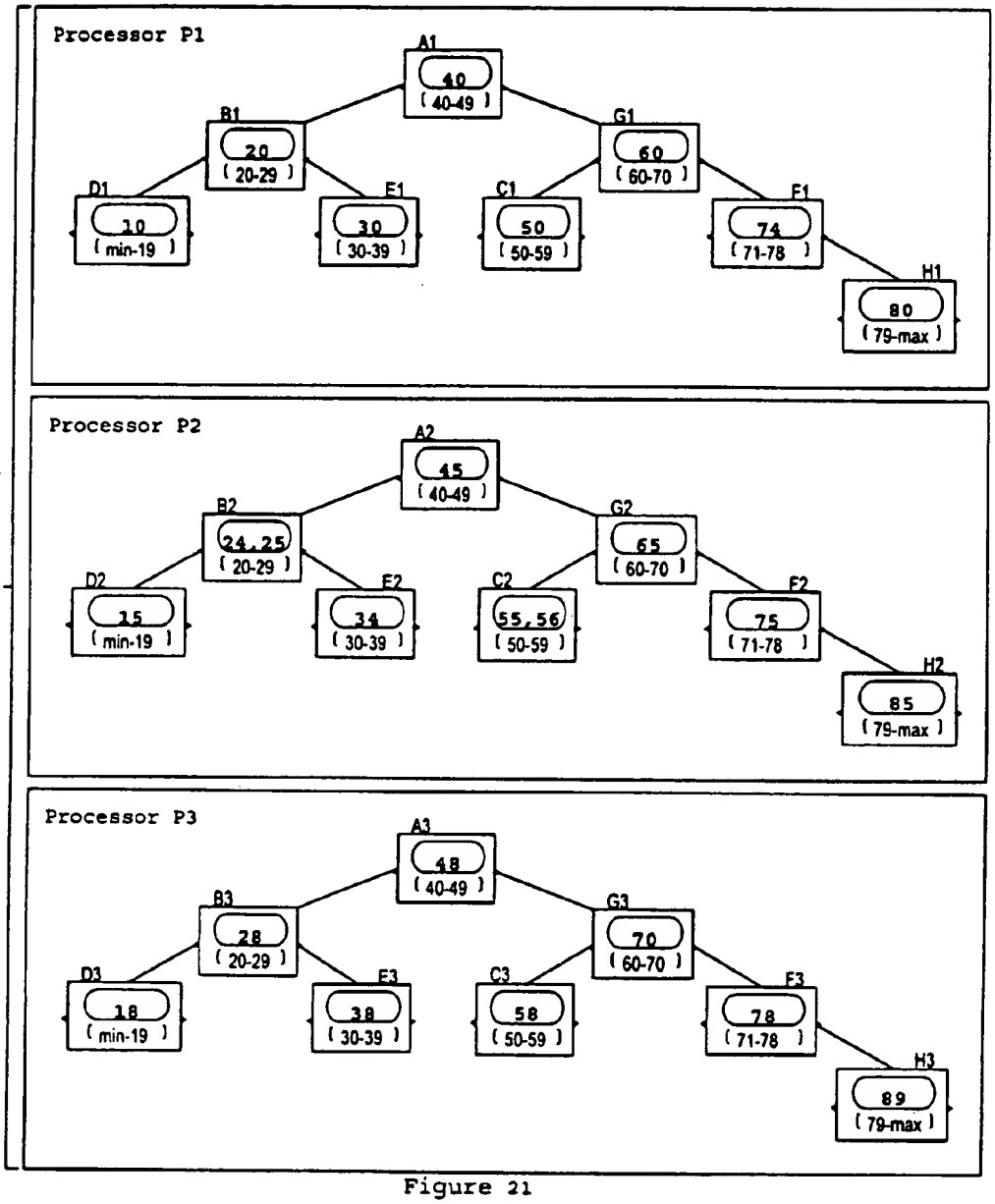



Fig 21

Figure 21

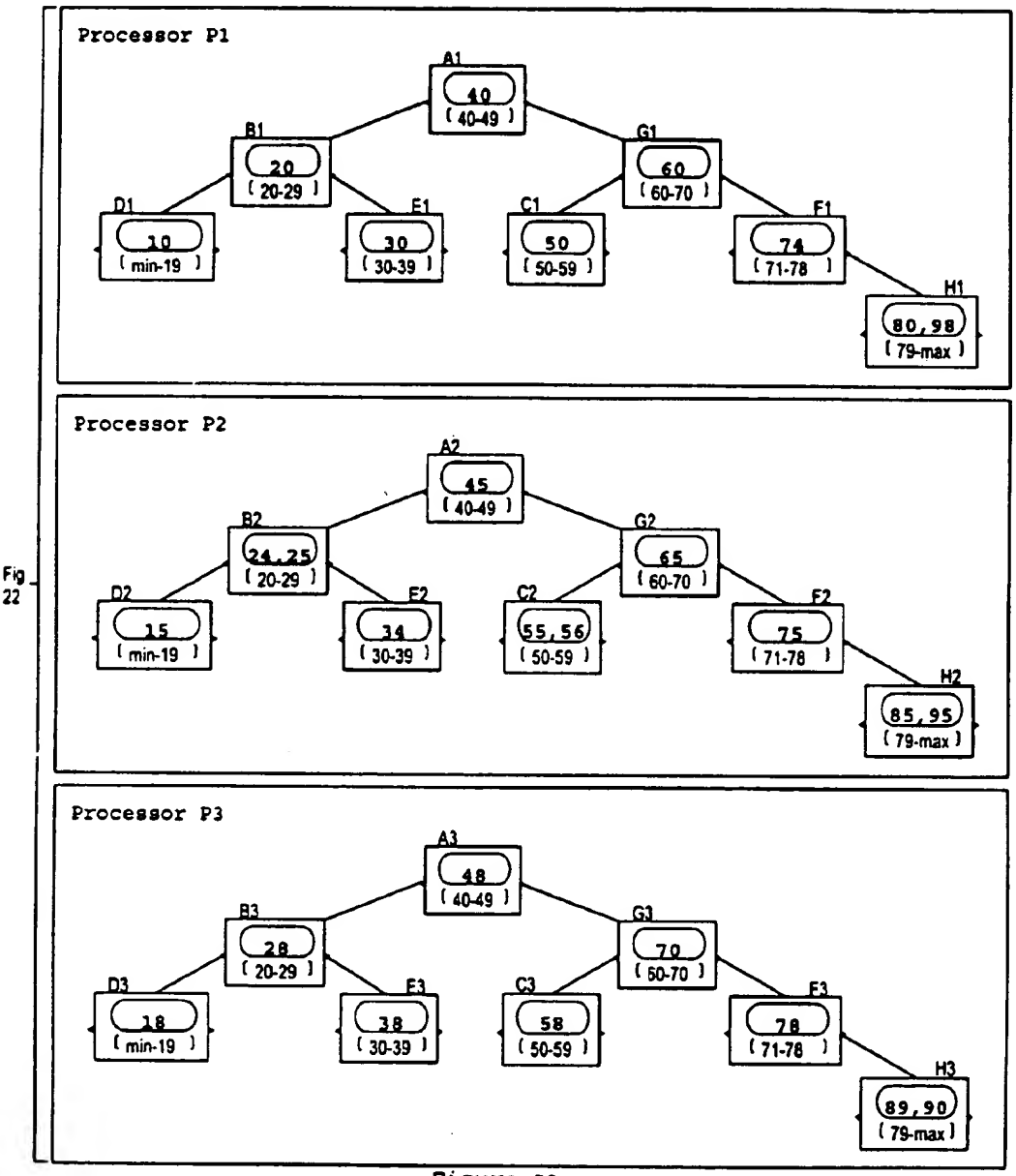



Fig 22

Figure 22

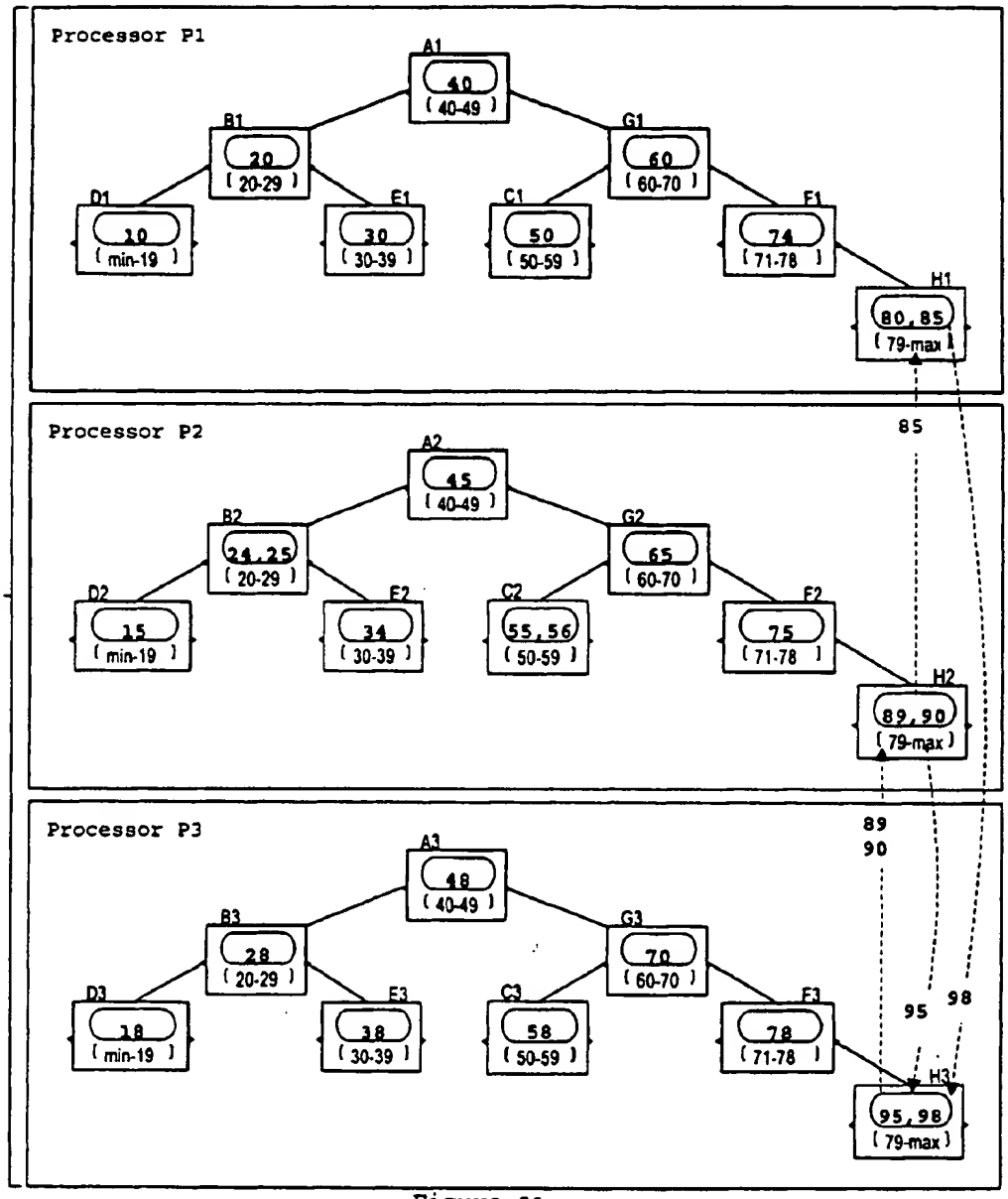



Fig 23

Figure 23

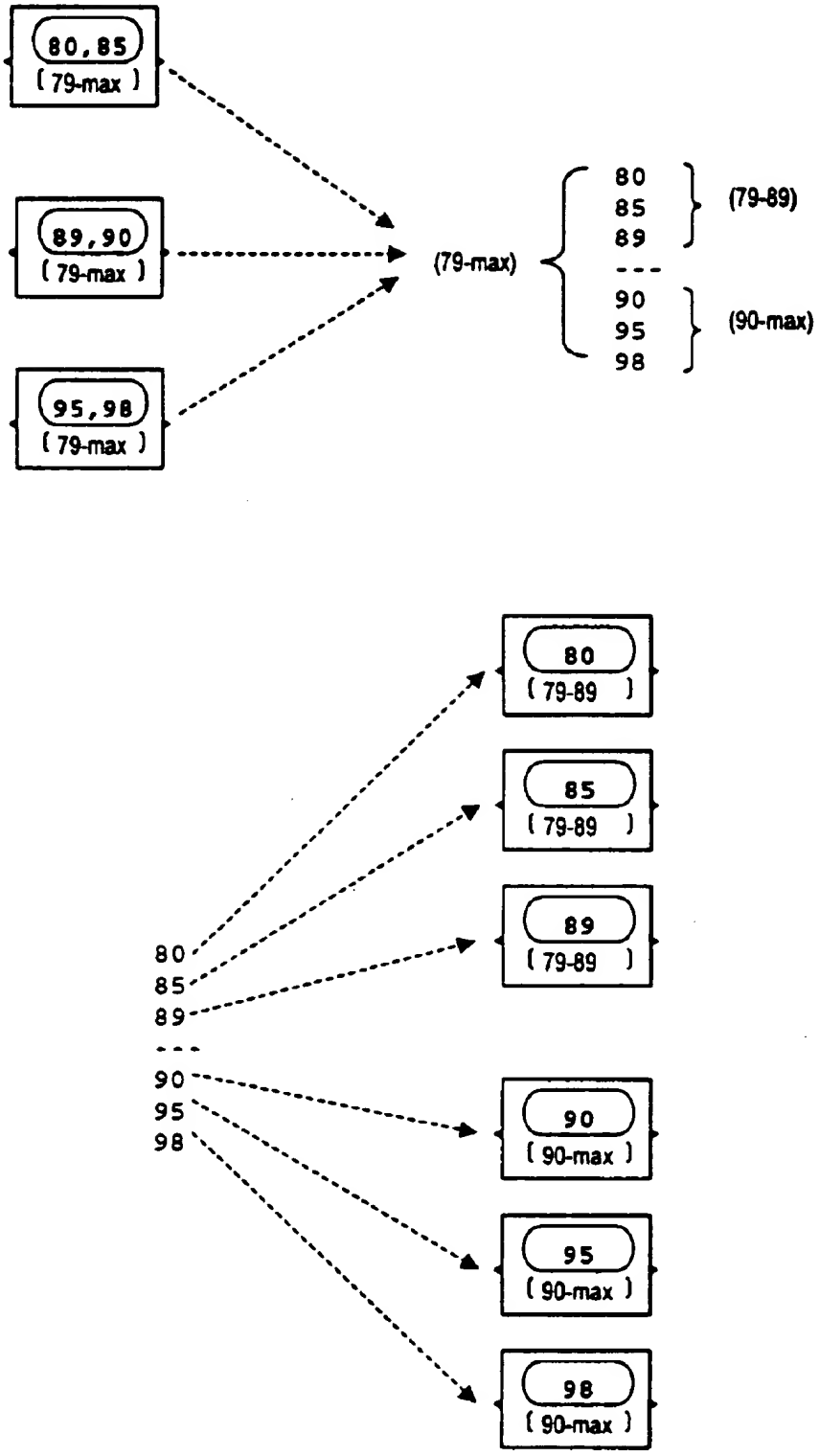

80,85
[ 79-max ]
89,90
[ 79-max ]
95,98
[ 79-max ]
(79-max)
80
85
89
---
90
95
98
(79-89)
(90-max)
80
[ 79-89 ]
85
[ 79-89 ]
89
[ 79-89 ]
90
[ 90-max ]
95
[ 90-max ]
98
[ 90-max ]

Figure 24

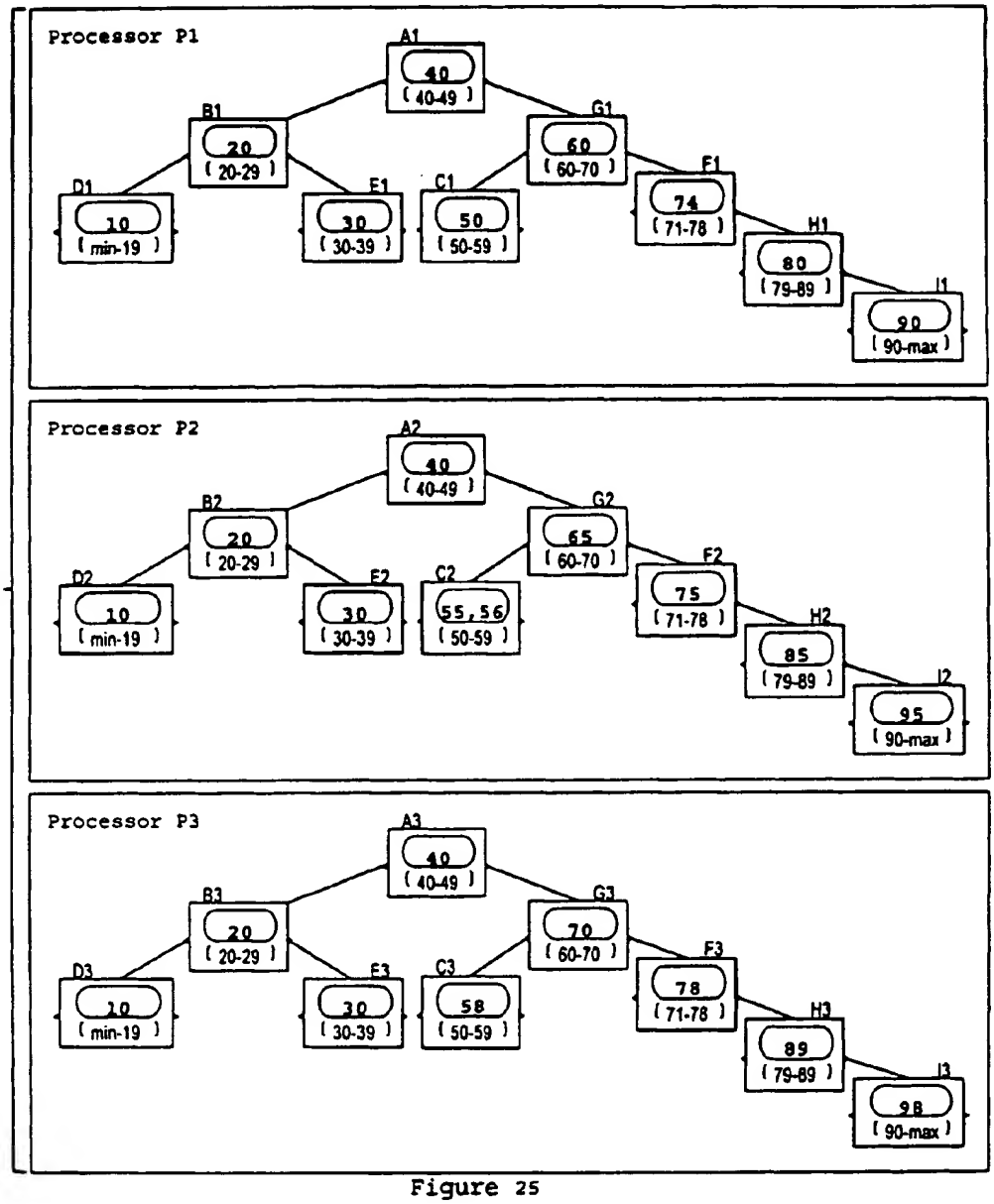



Fig 25

Figure 25

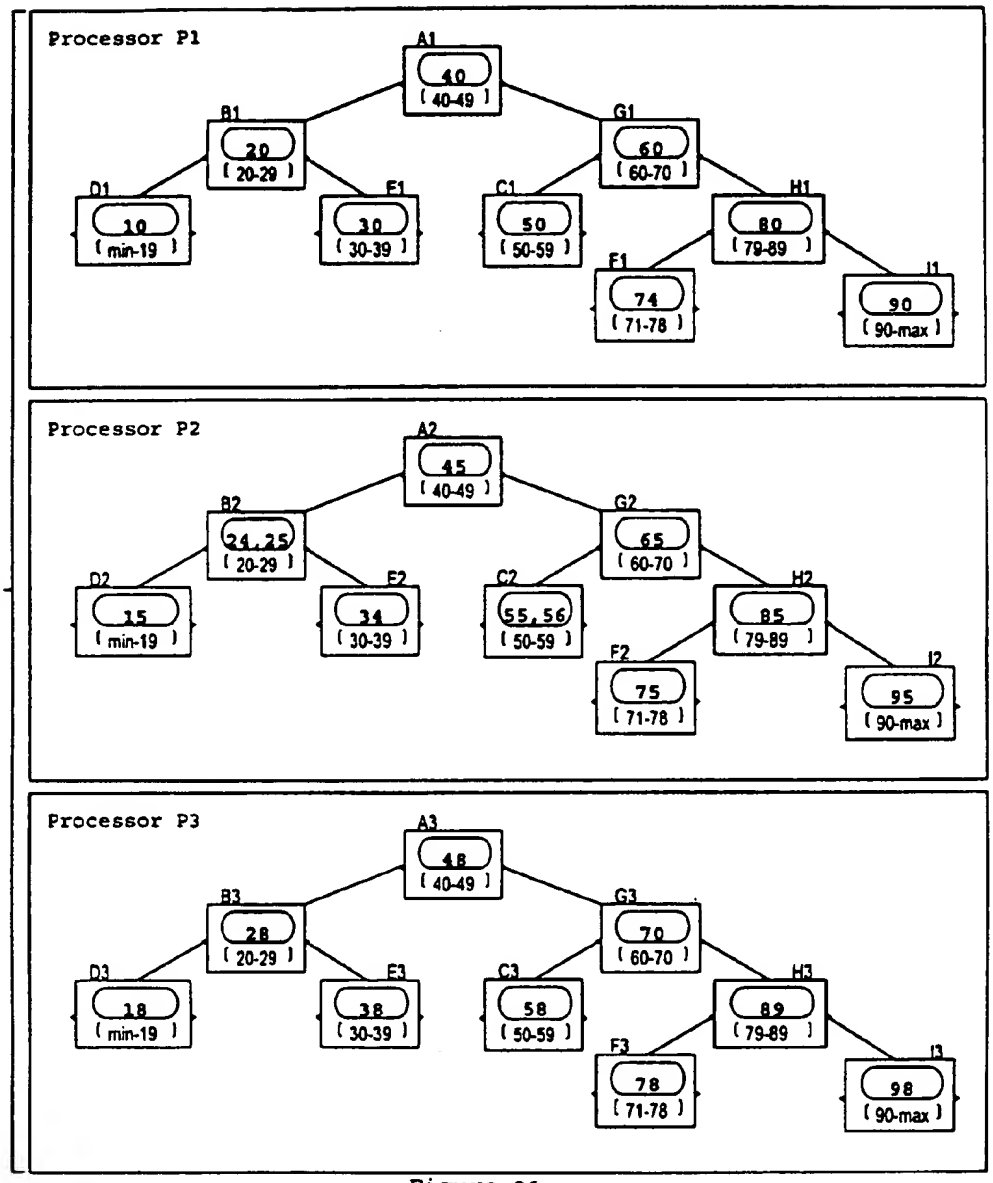



Fig 26

Figure 26

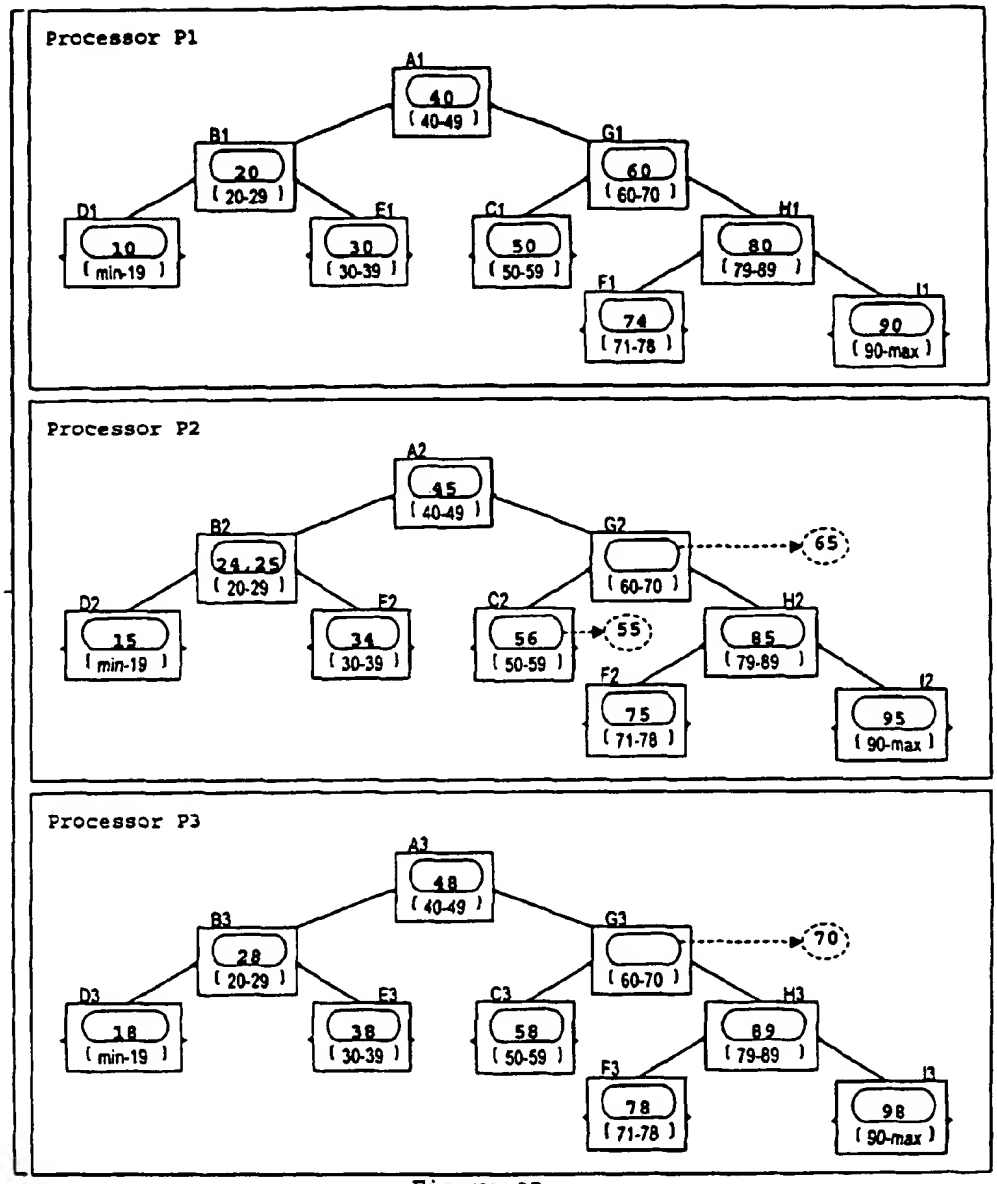



Fig 27

Figure 27

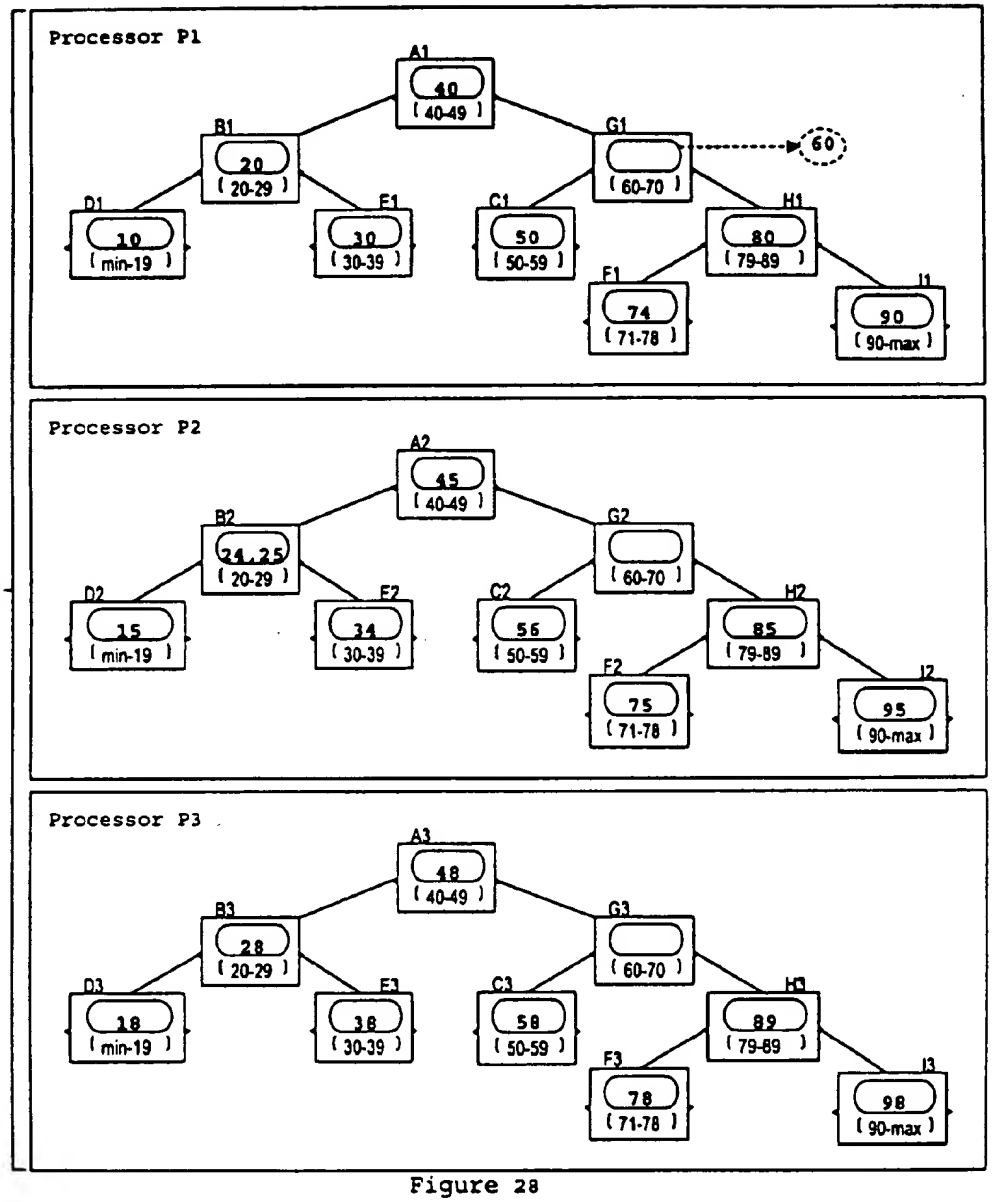



Fig 28

Figure 28

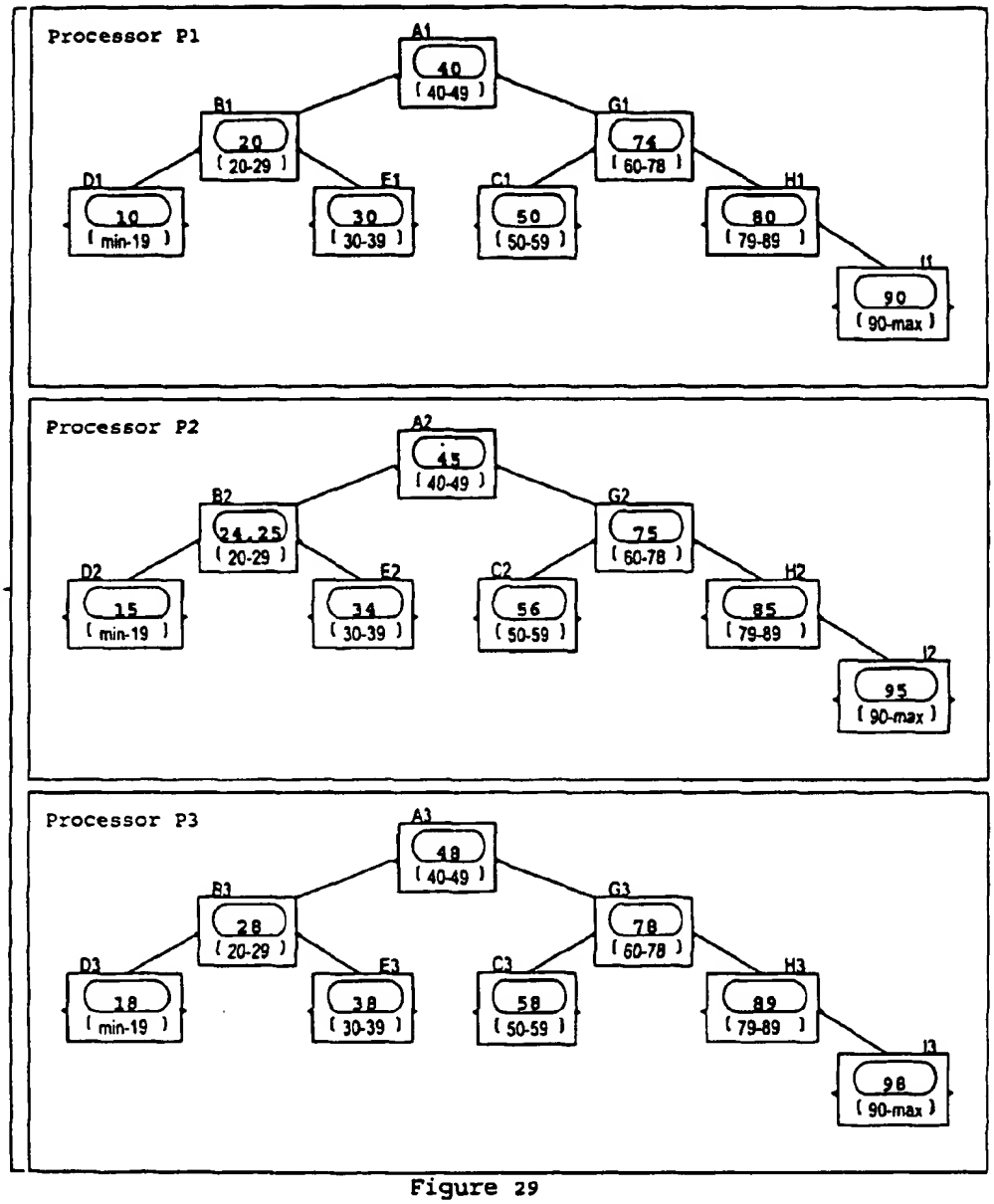



Fig 29

Figure 29

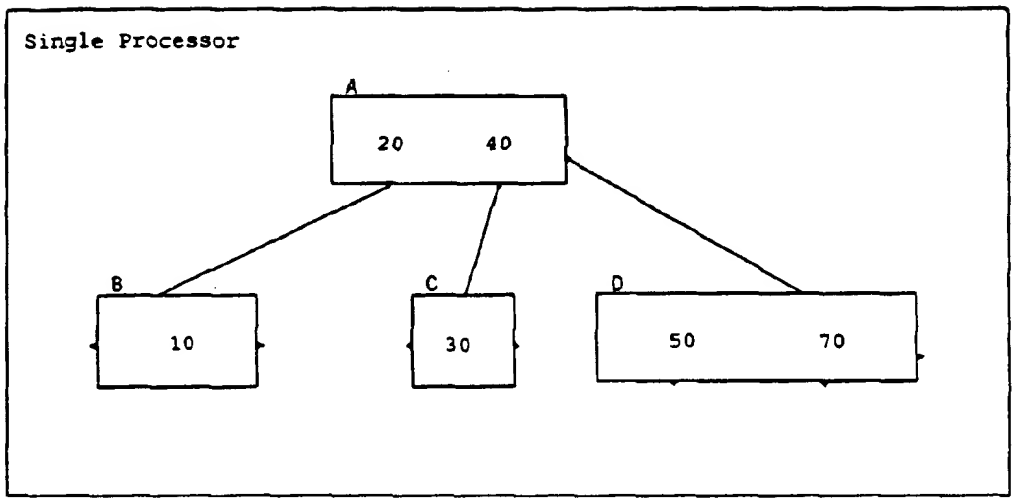



Figure 30

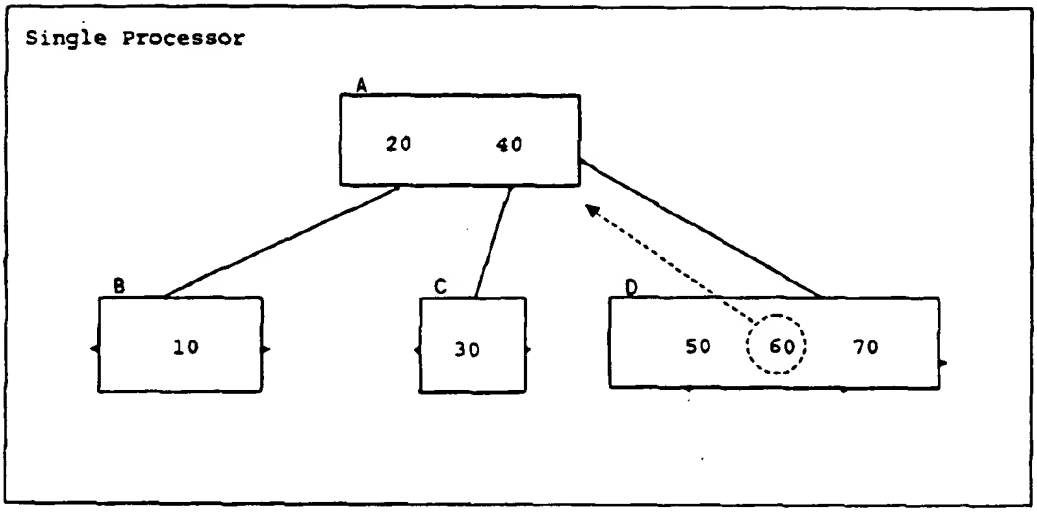



Figure 31

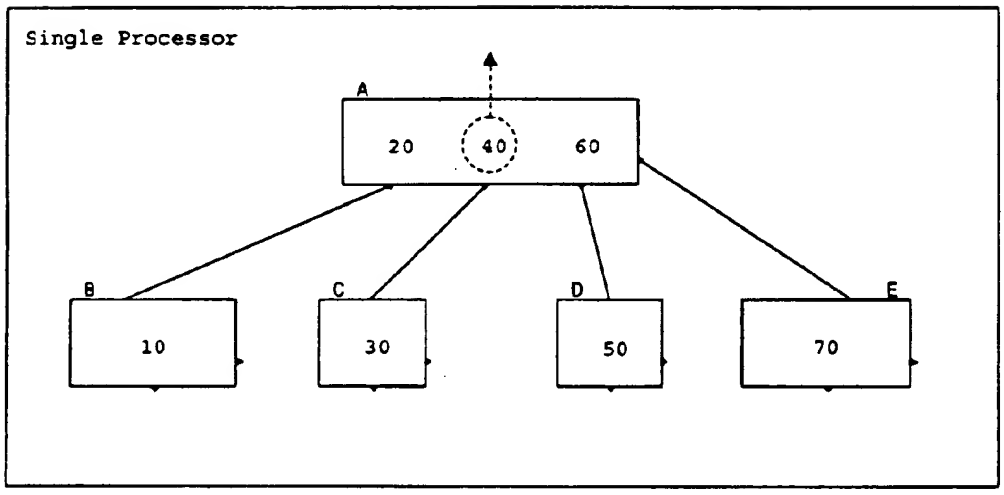



Figure 32

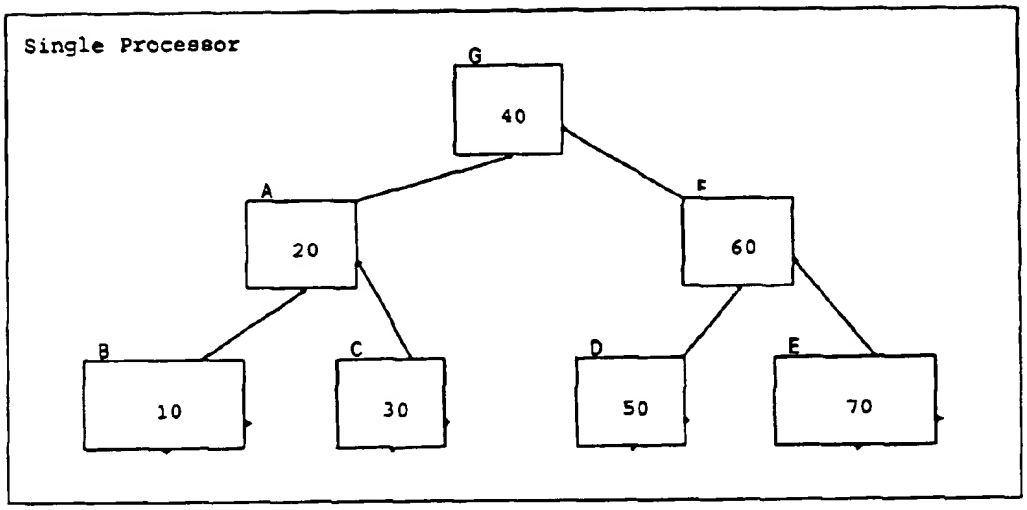



Figure 33

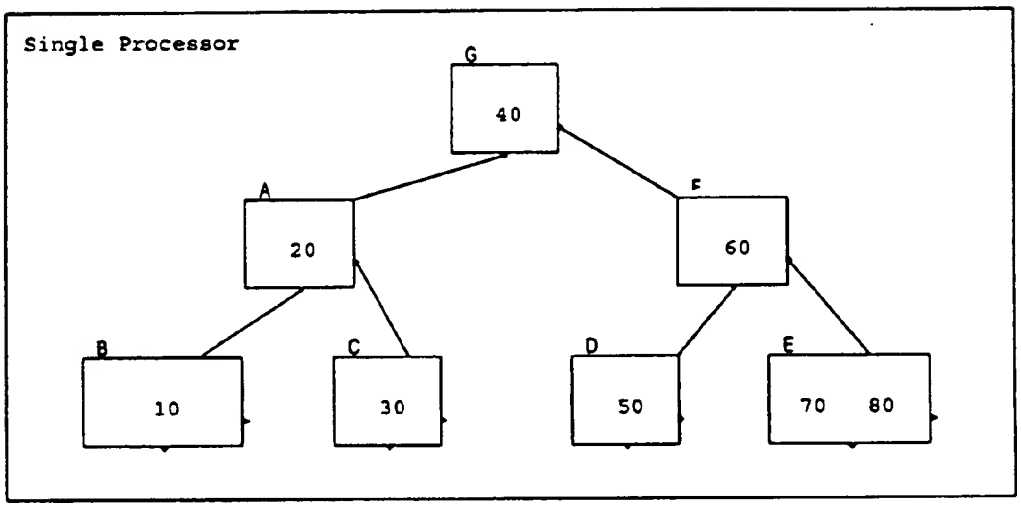



Figure 34

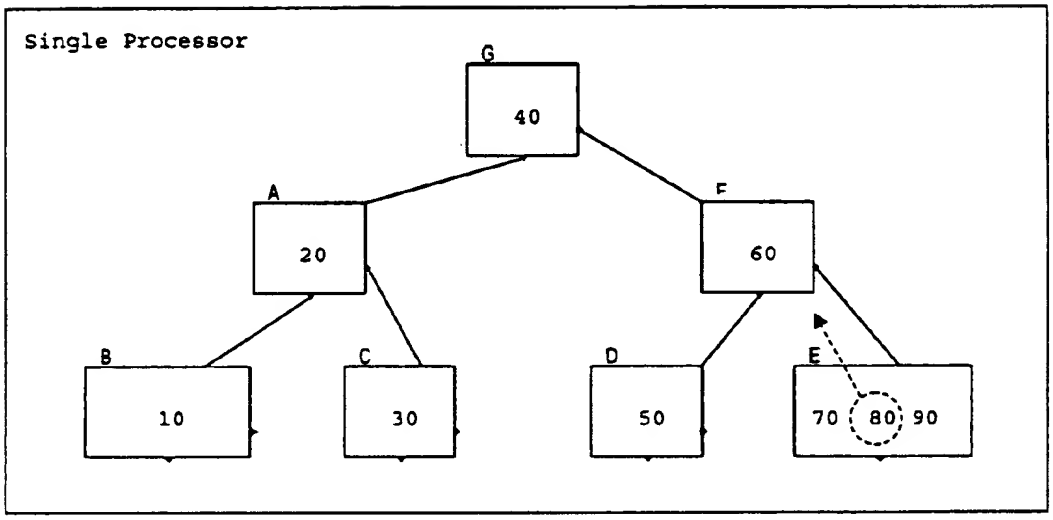



Figure 35

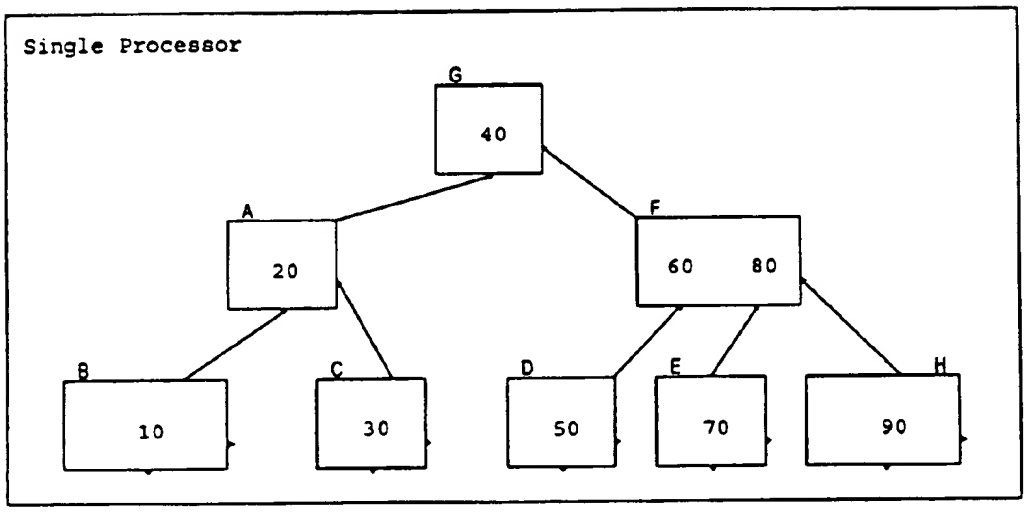



Figure 36

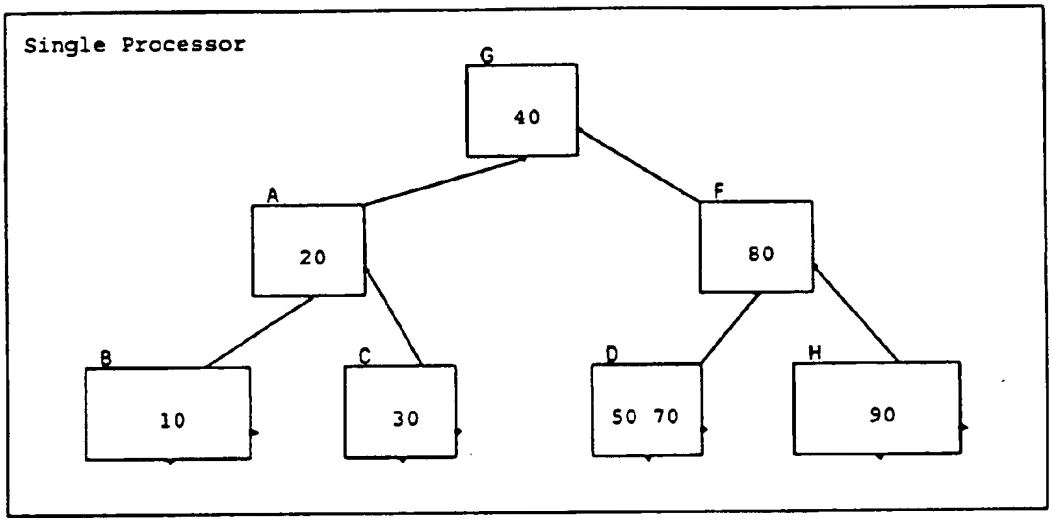



Figure 37

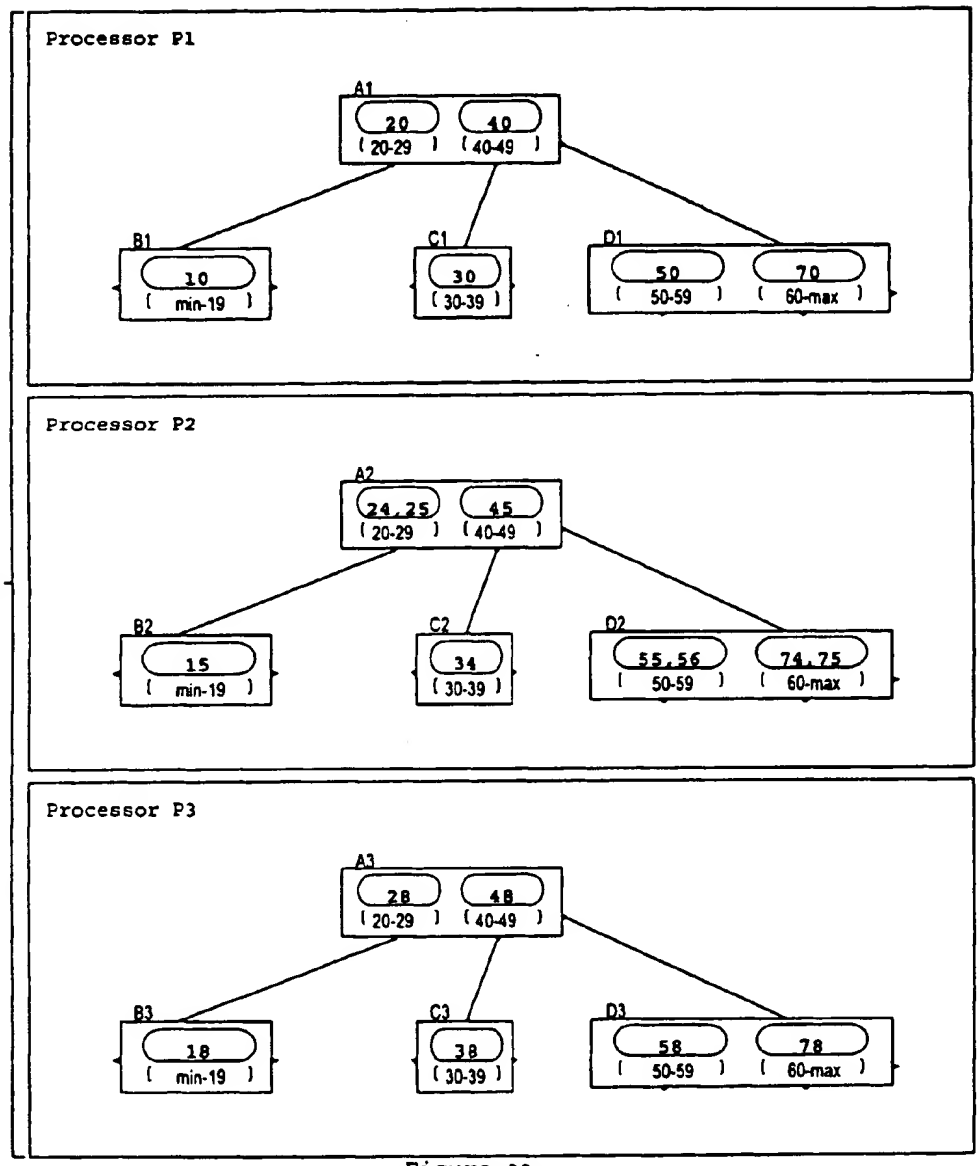



Fig 38

Figure 38

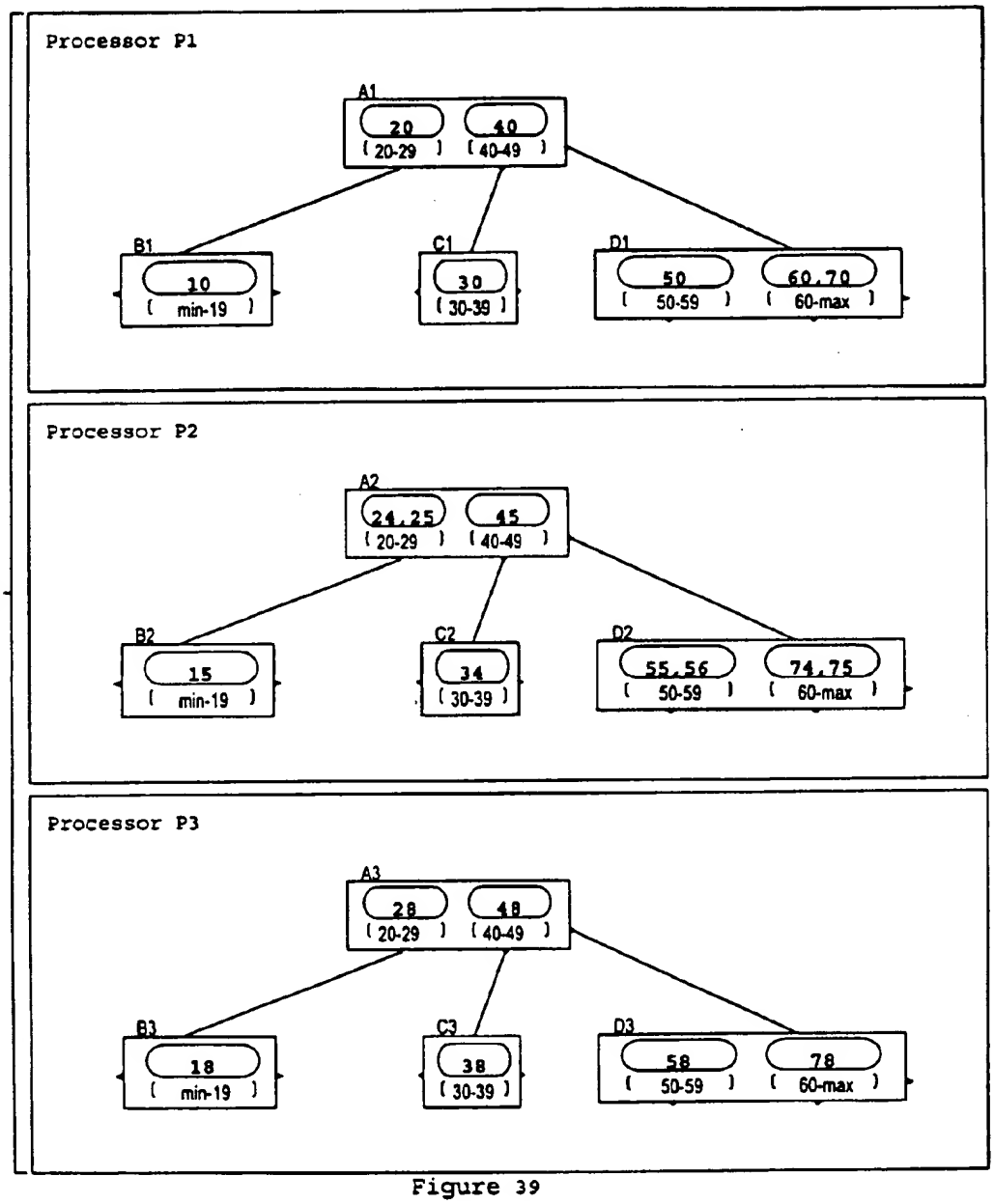



Fig 39

Figure 39

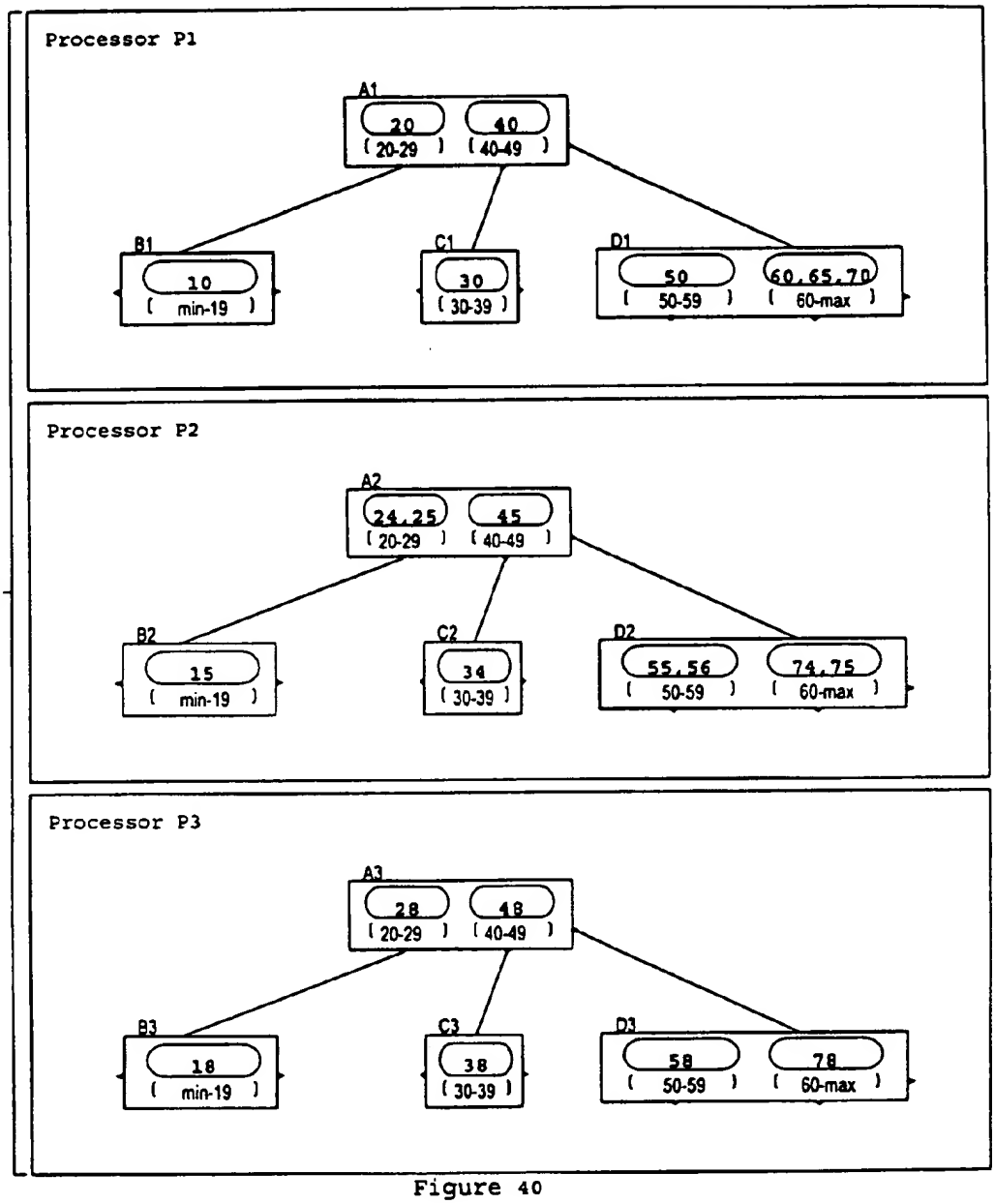



Fig 40

Figure 40

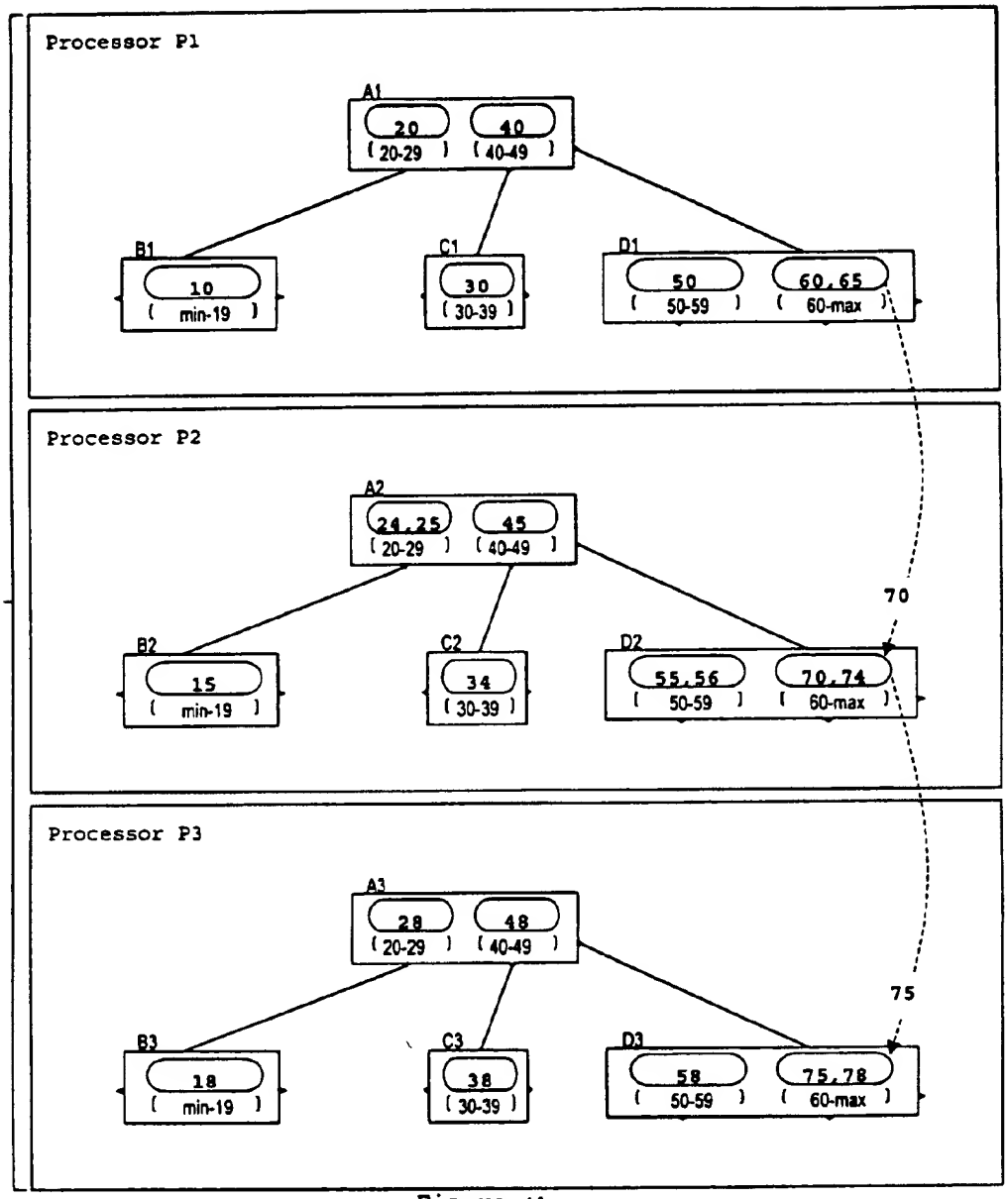



Fig 41

Figure 41

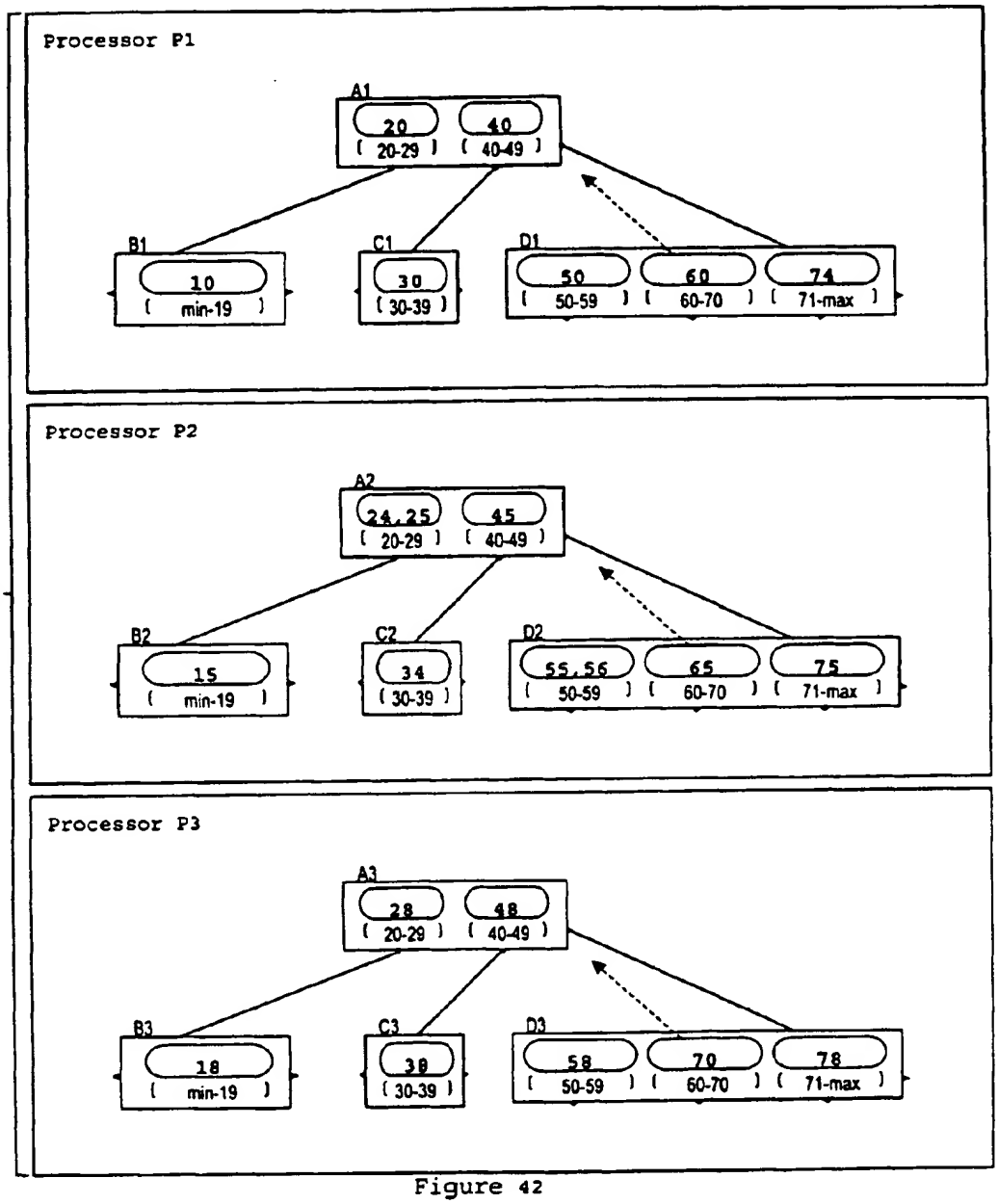



Fig 42

Figure 42

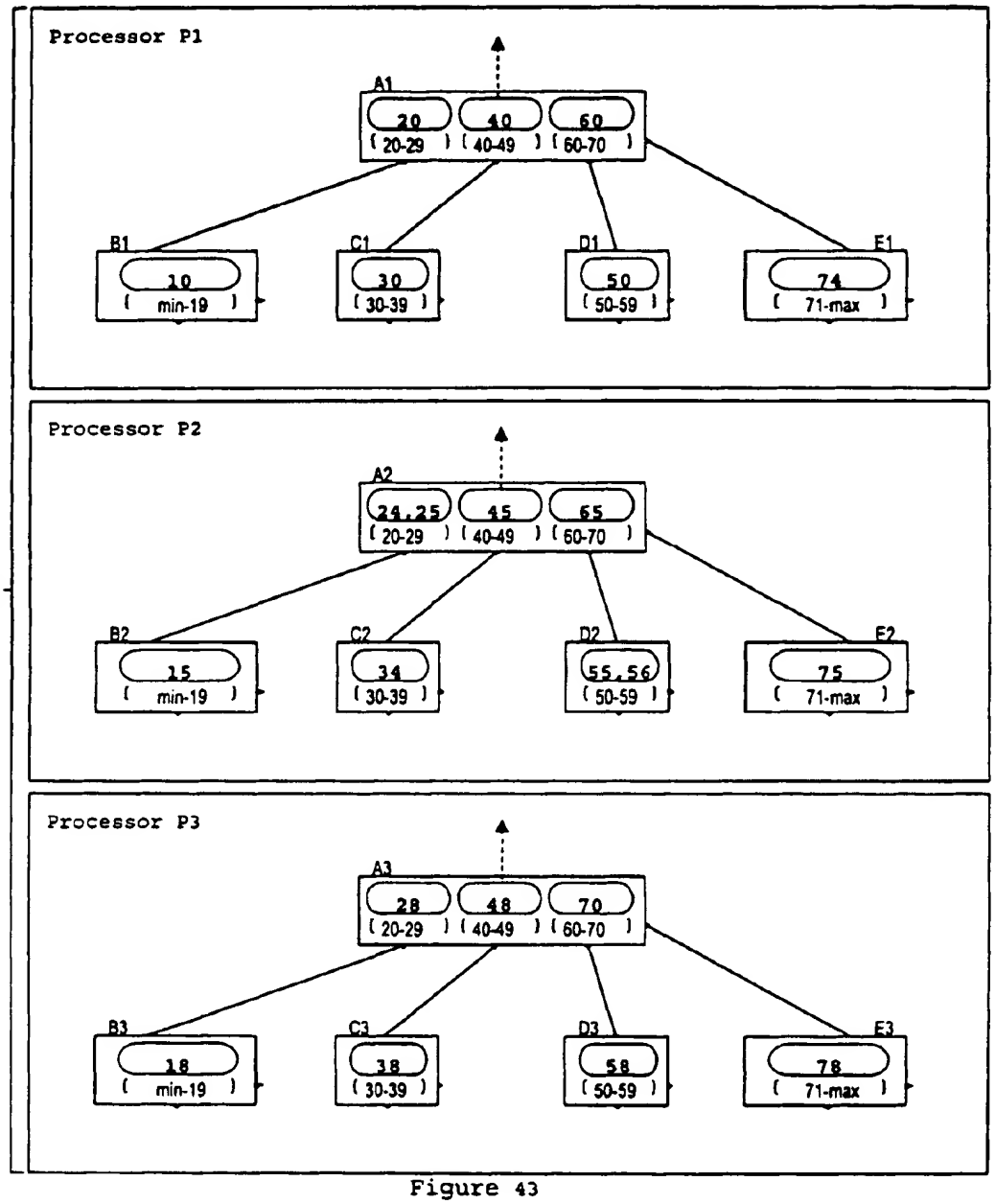



Fig 43

Figure 43

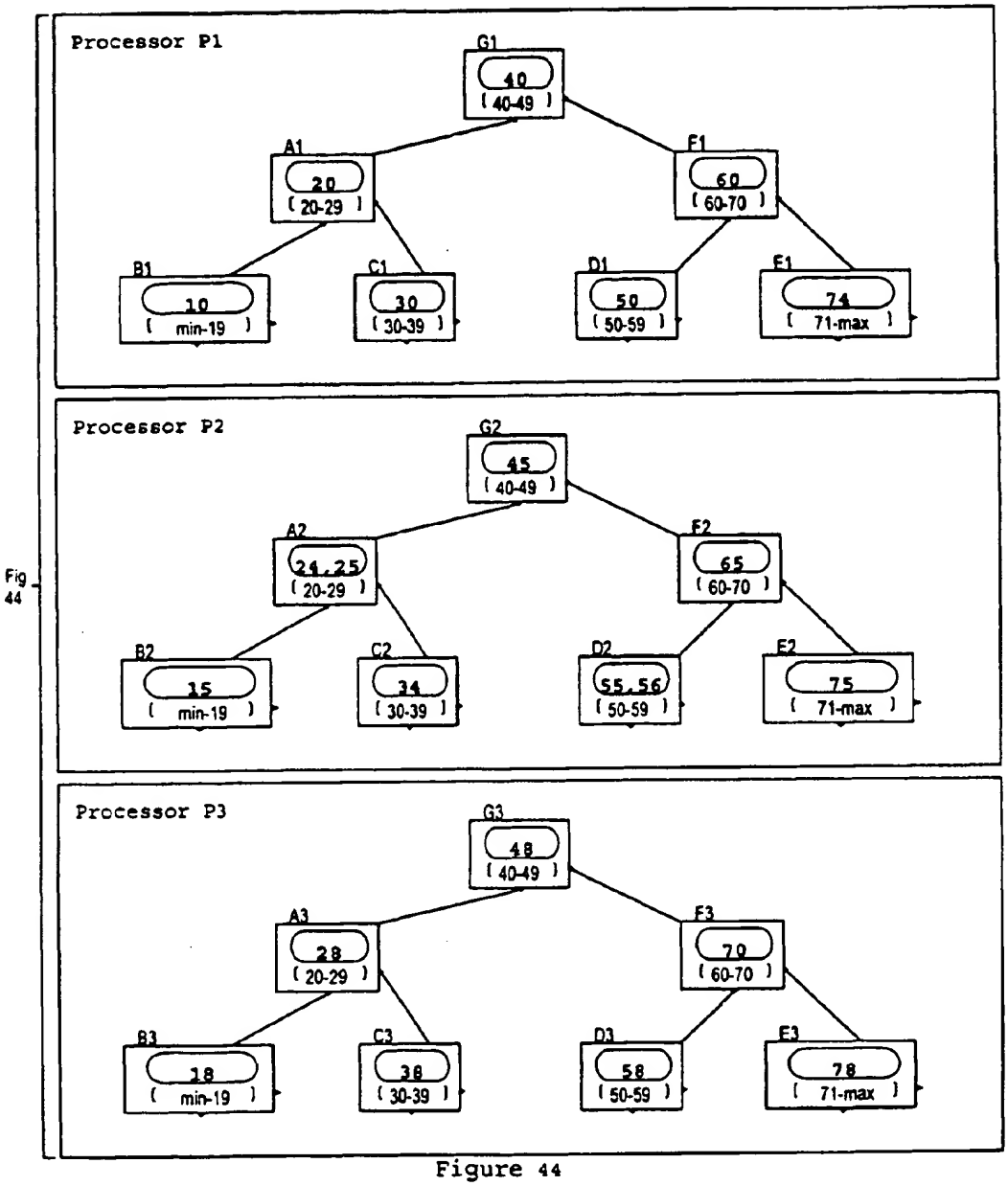



Fig 44

Figure 44

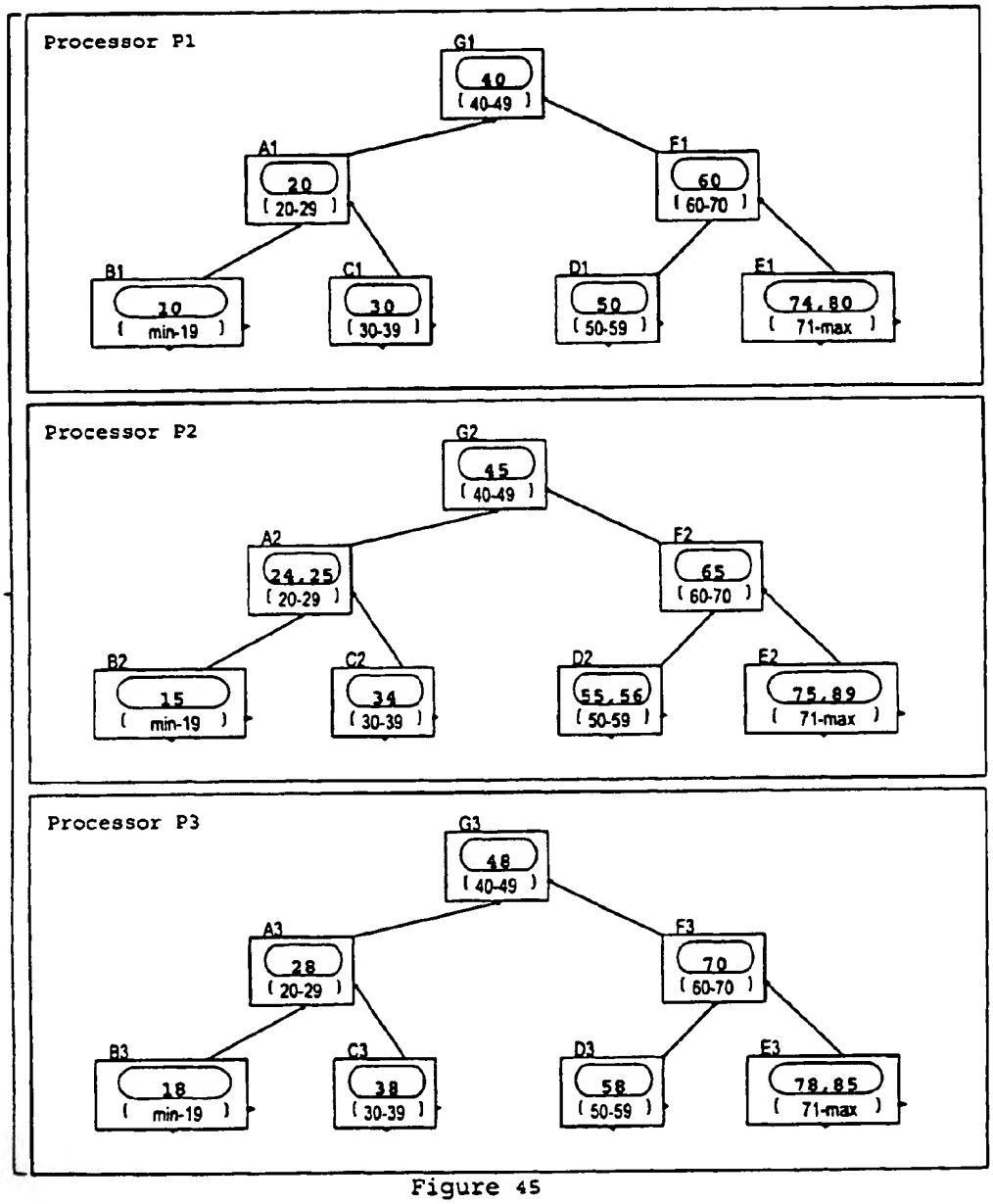



Fig 45

Figure 45

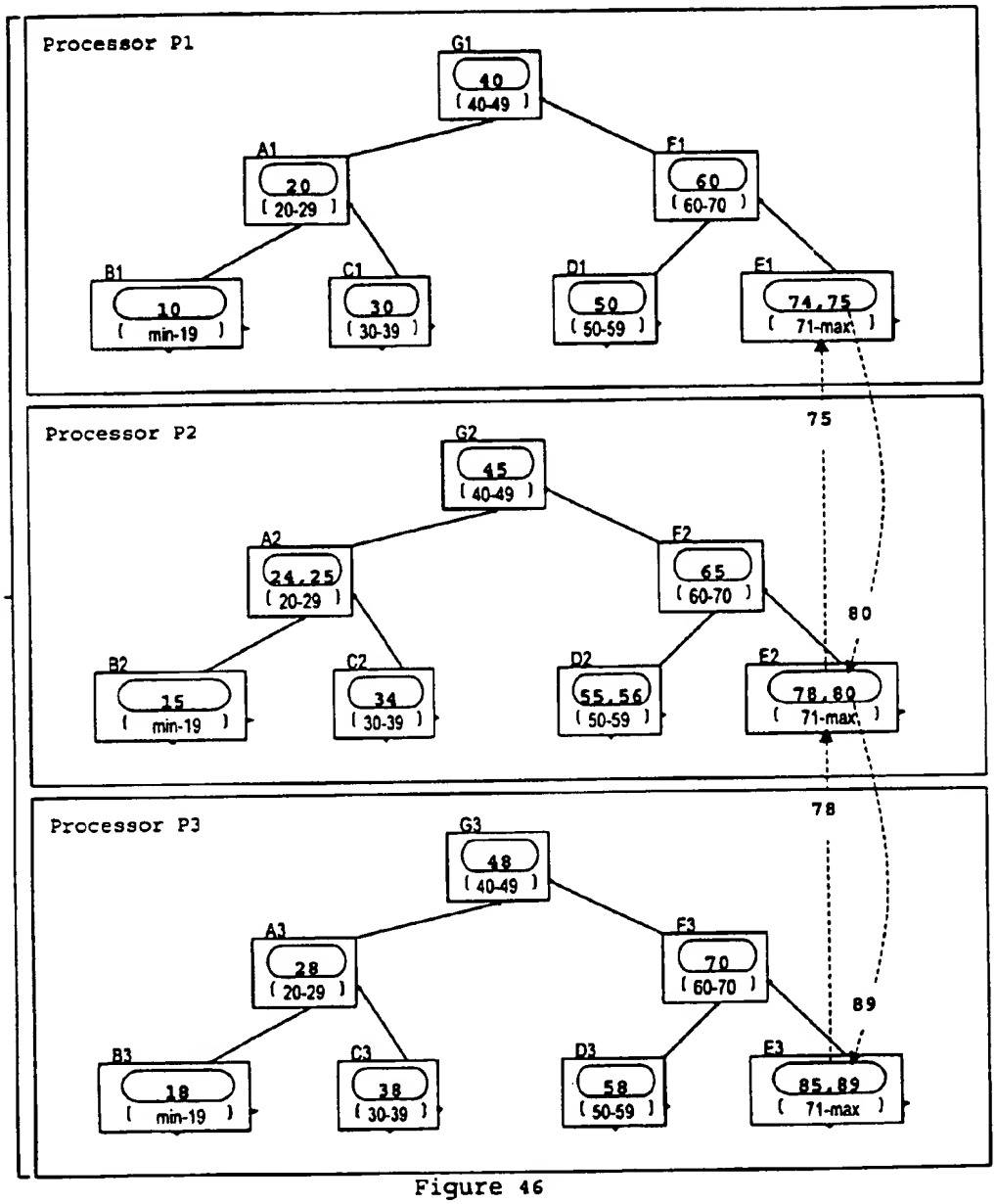



Fig 46

Figure 46

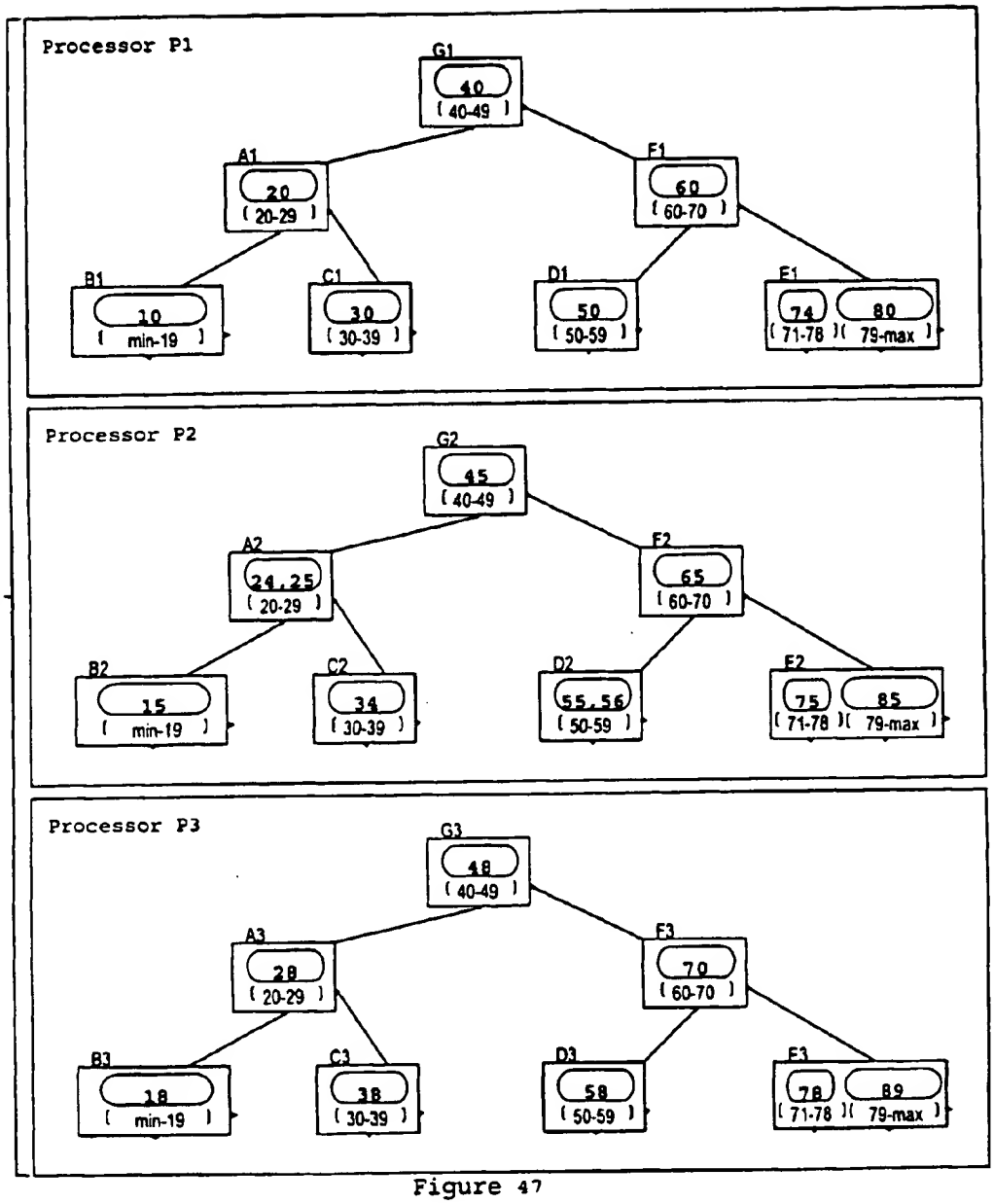



Fig 47

Figure 47

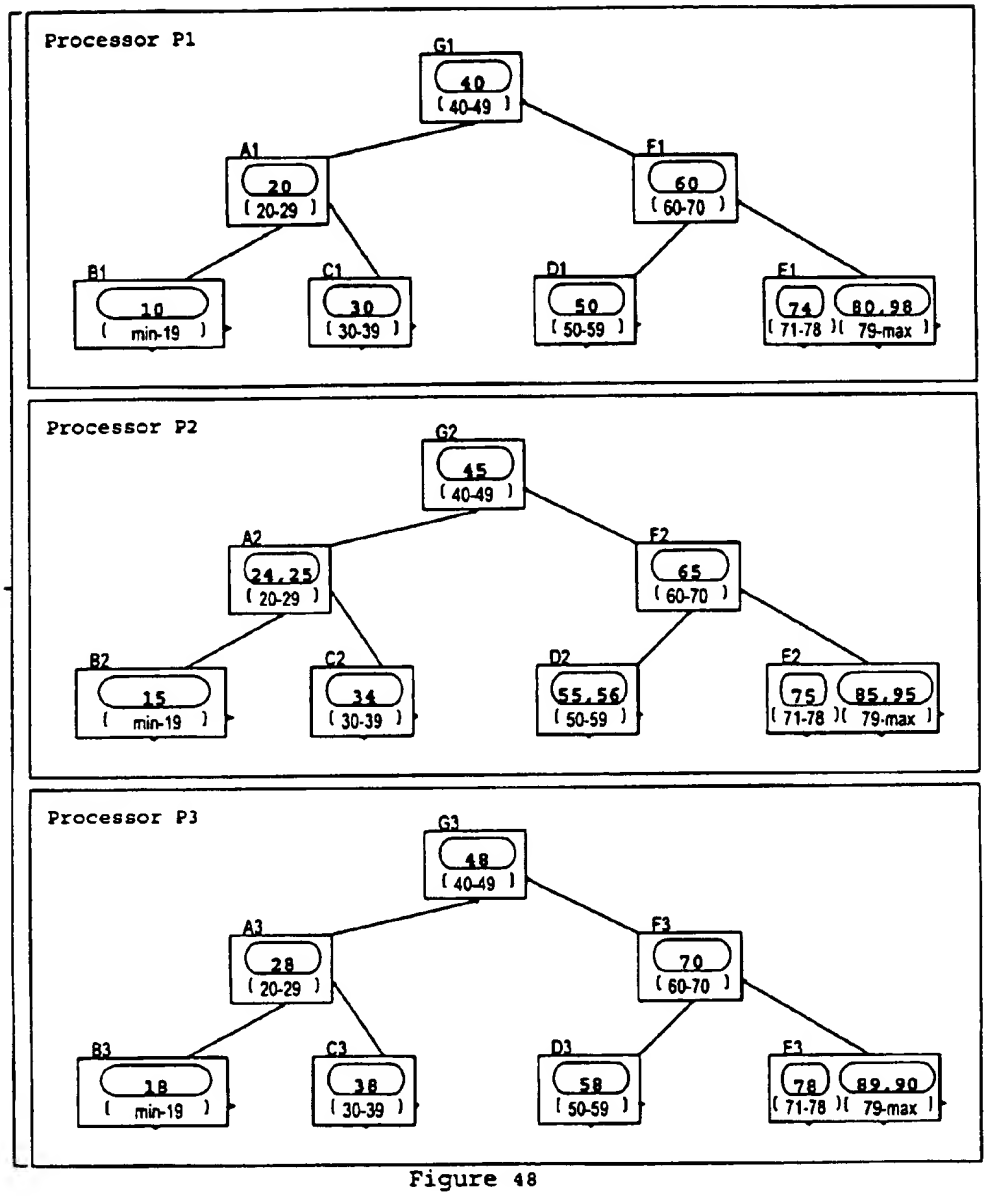



Fig 48

Figure 48

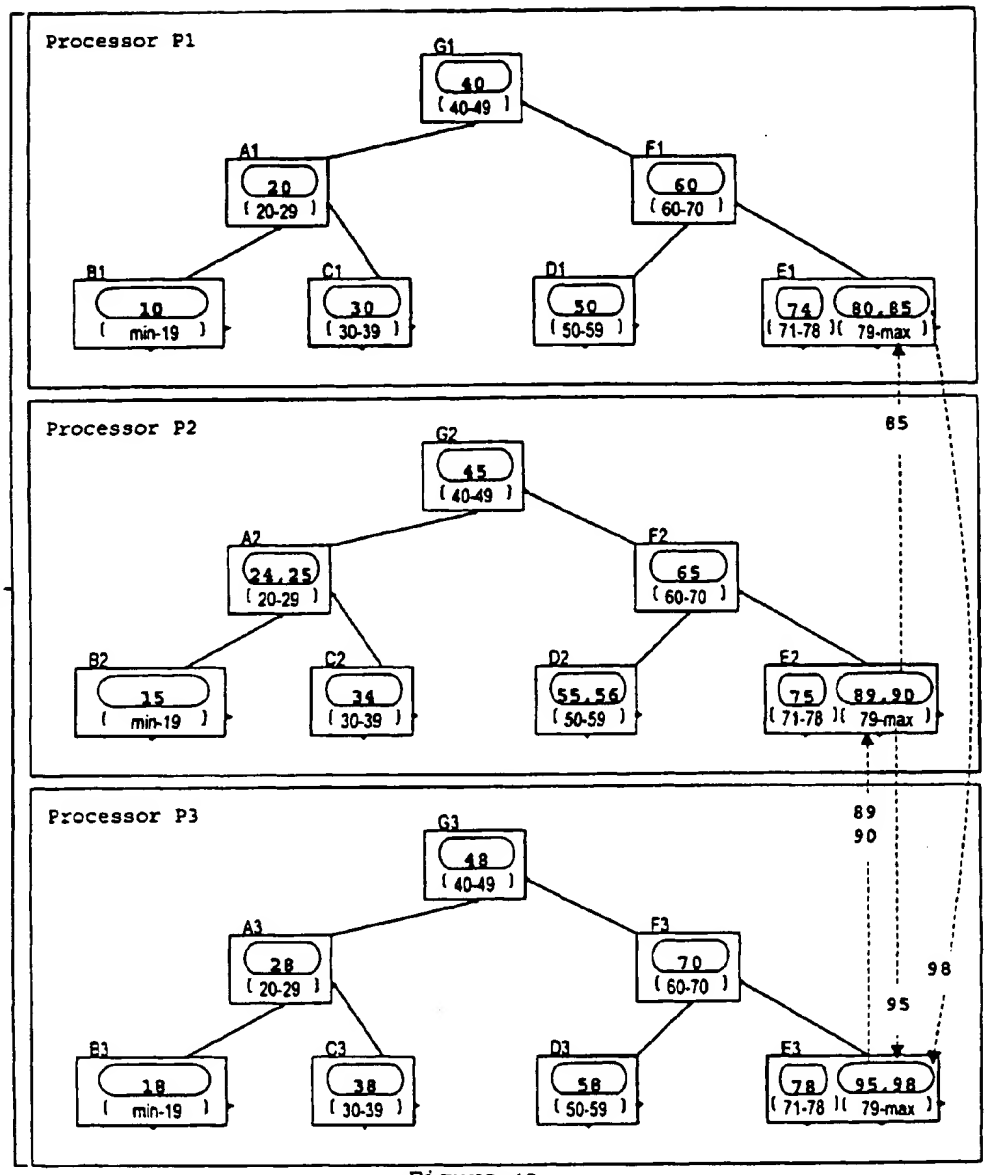



Fig 49

Figure 49

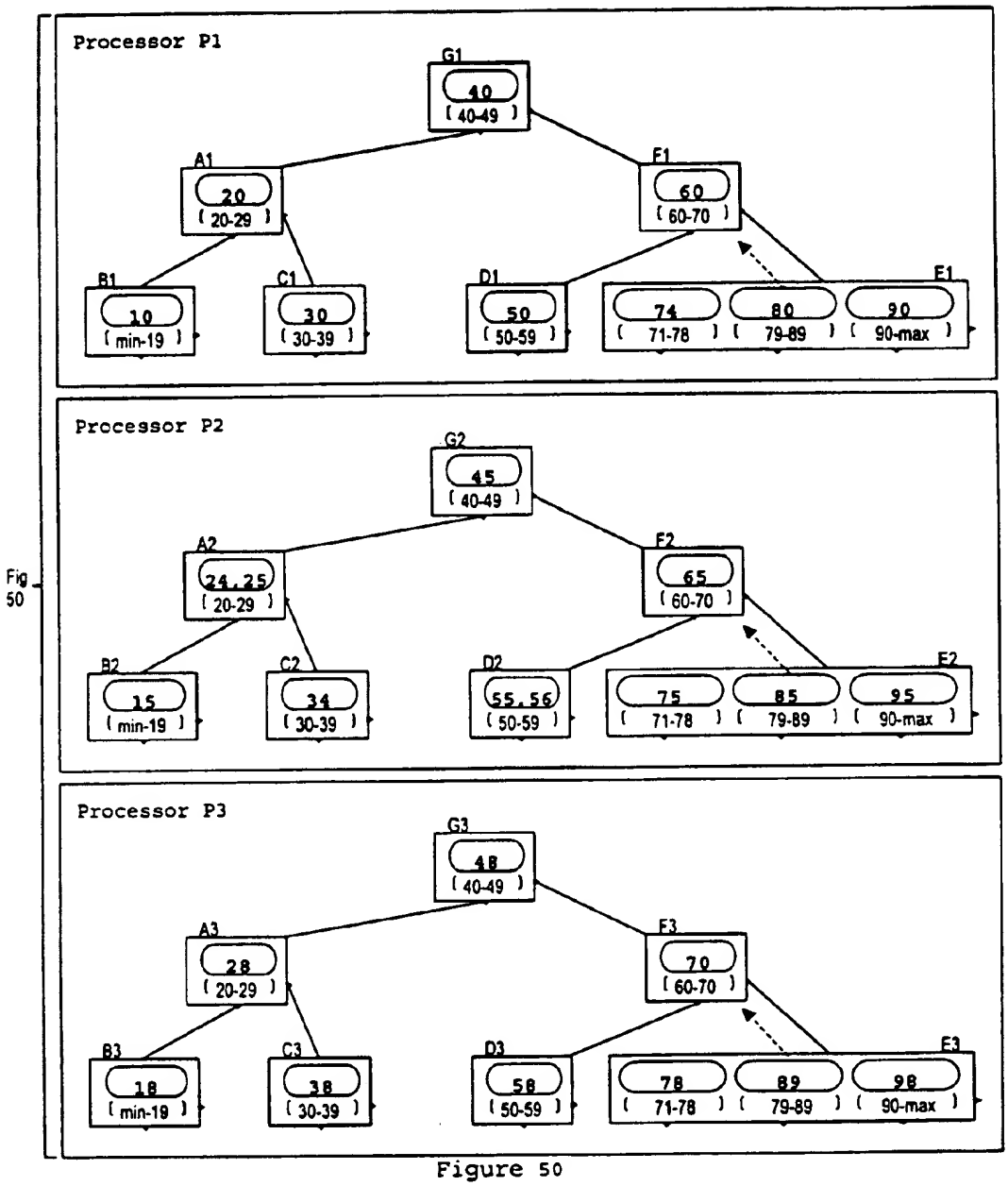



Fig 50

Figure 50

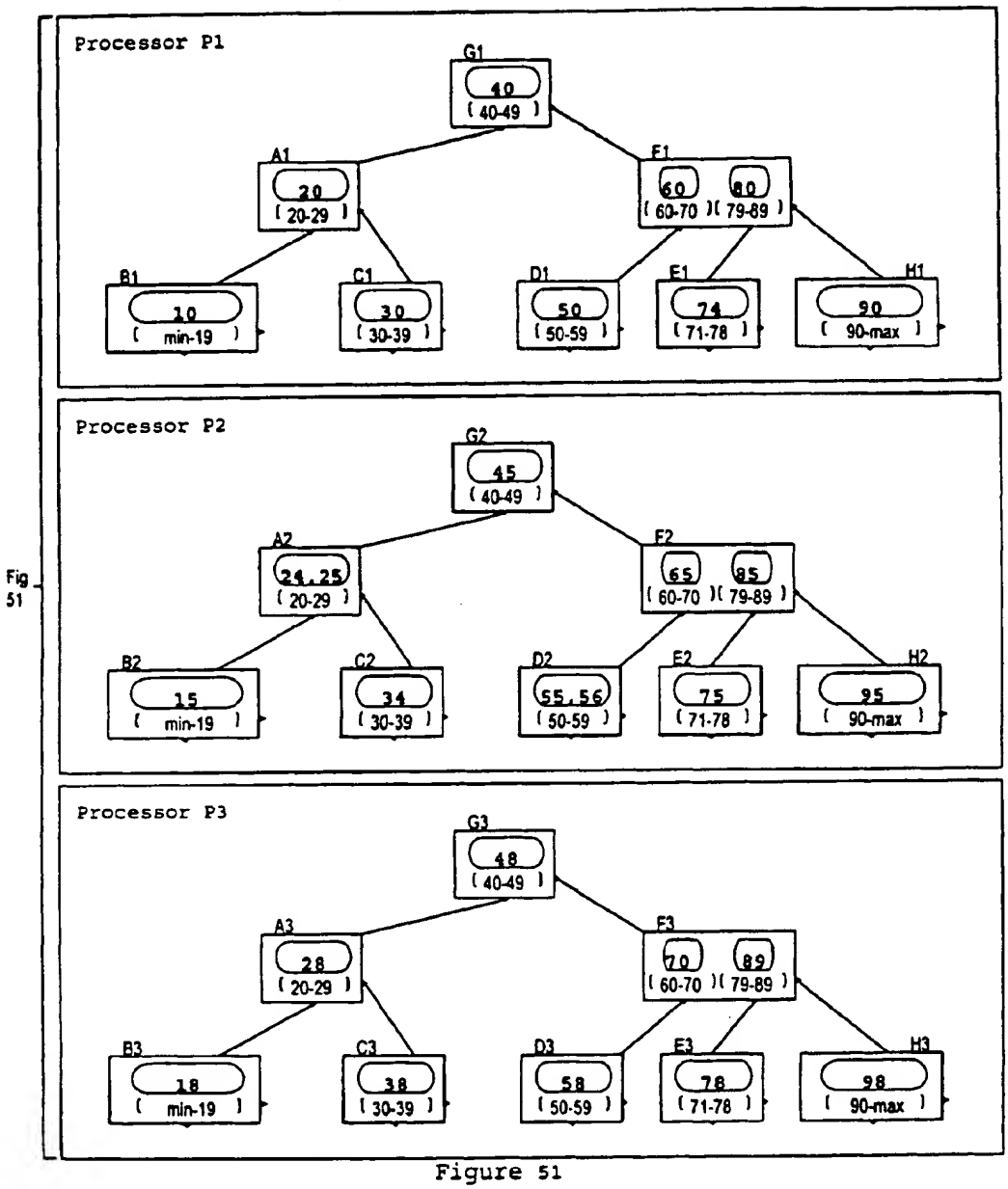



Fig 51

Figure 51

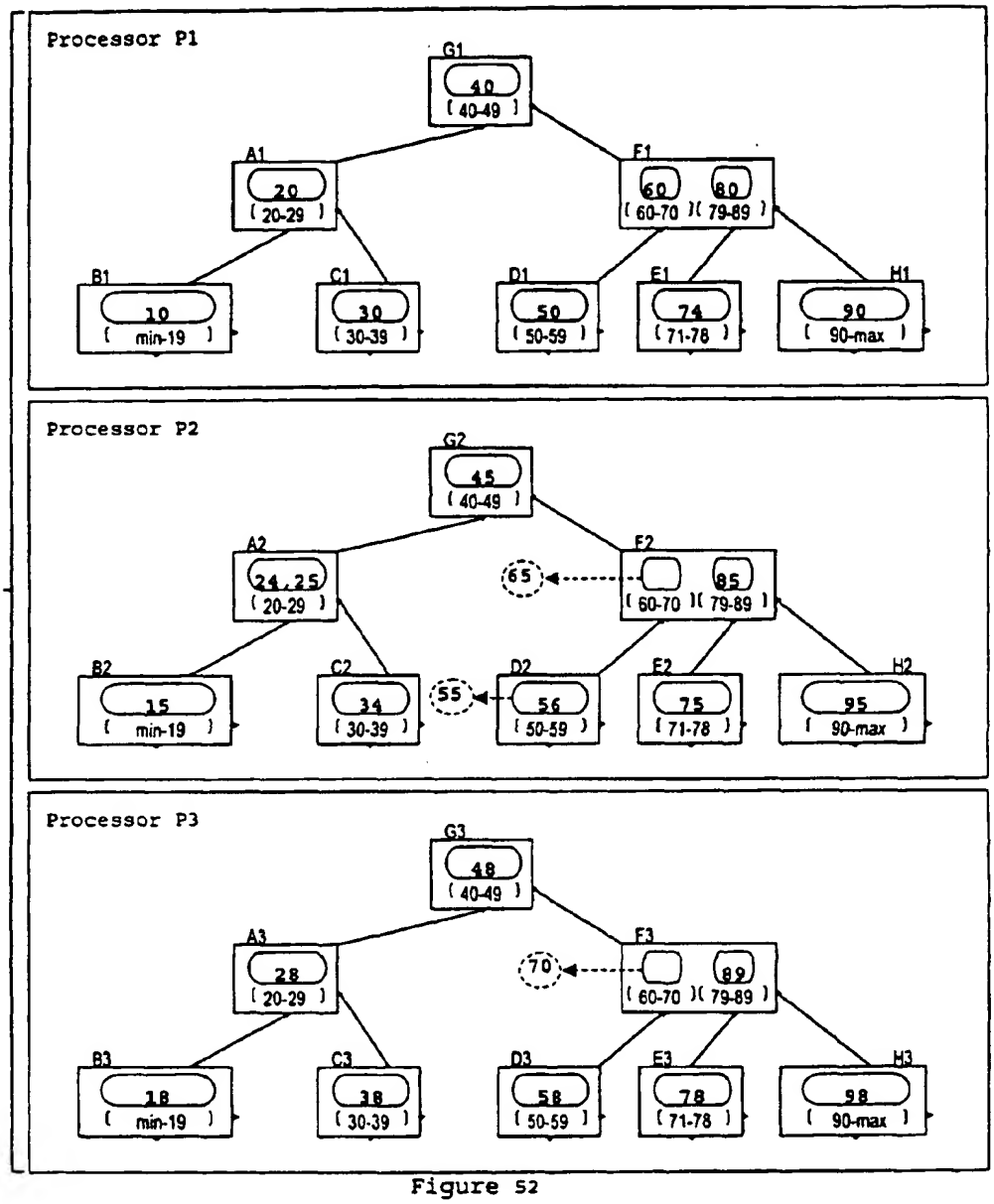



Fig 52

Figure 52

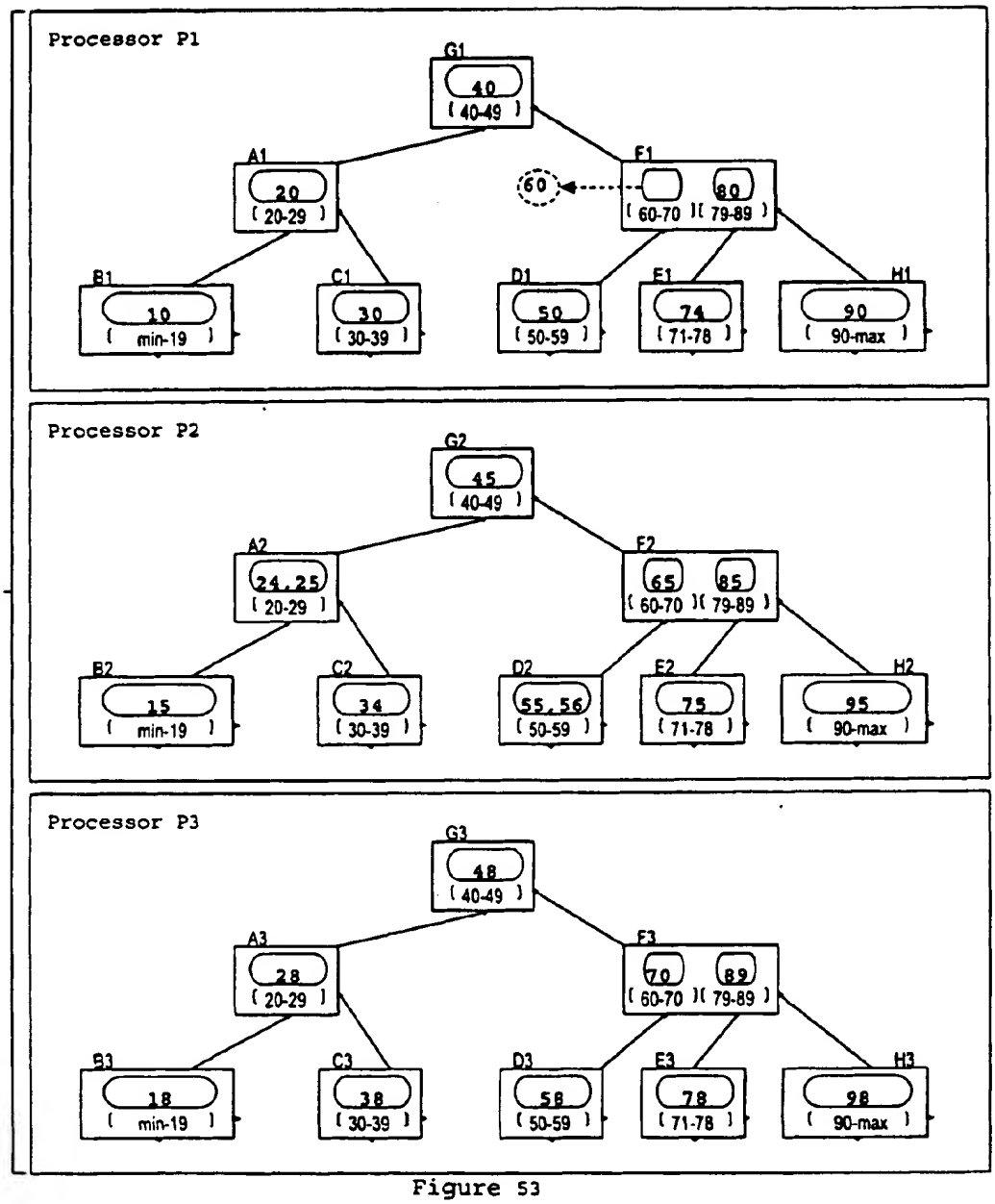



Fig 53

Figure 53

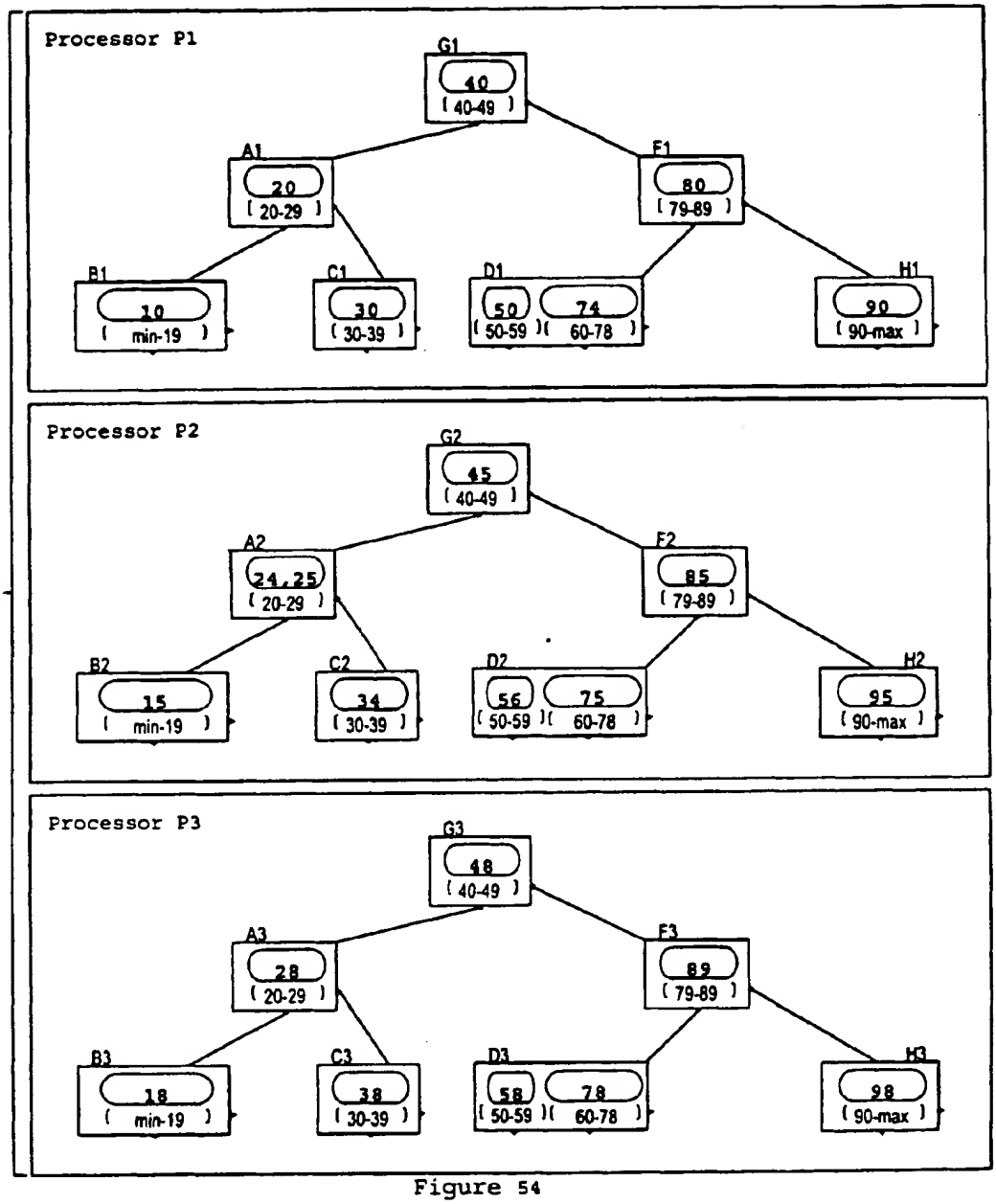



Fig 54

Figure 54

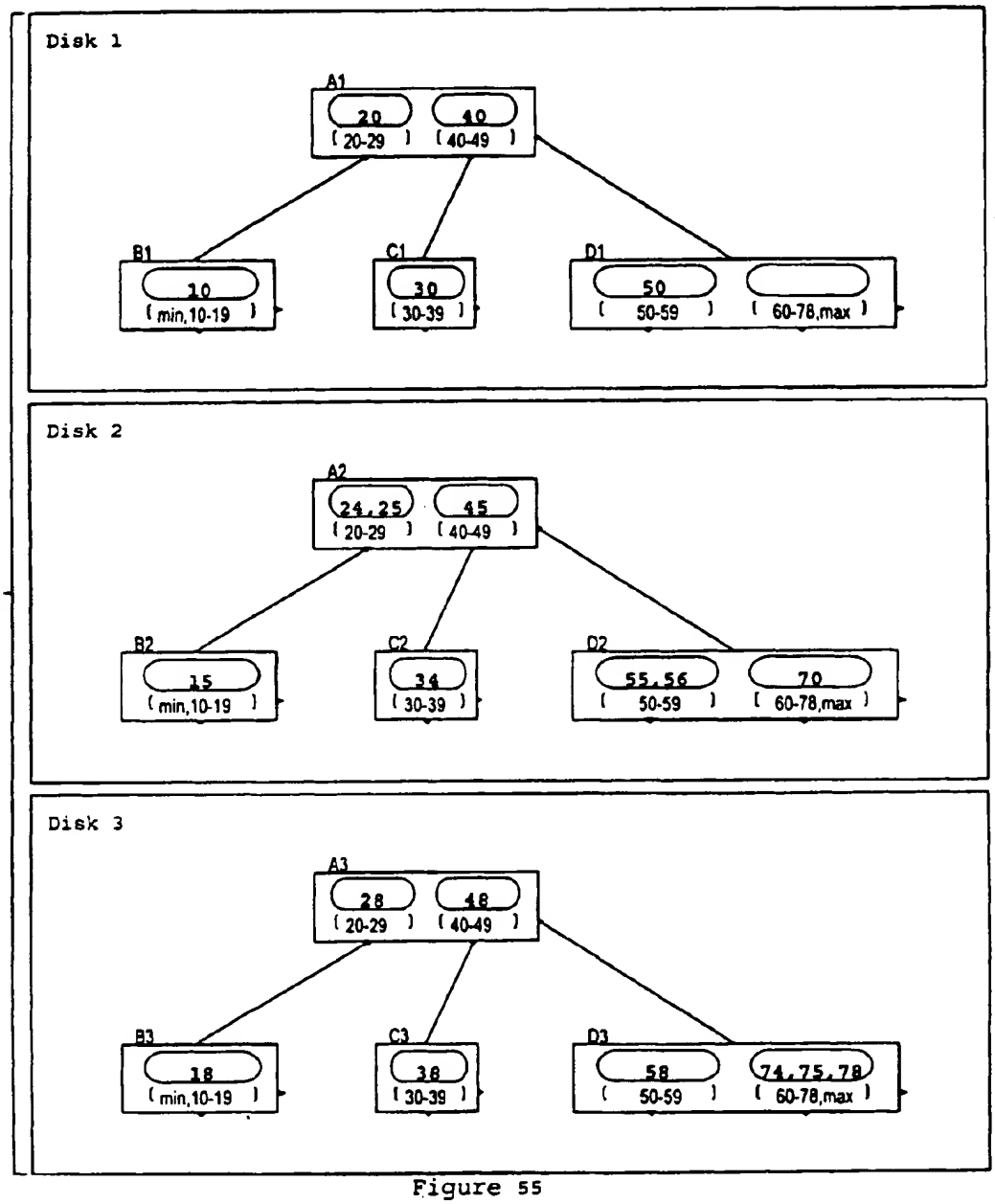



Fig 55

Figure 55

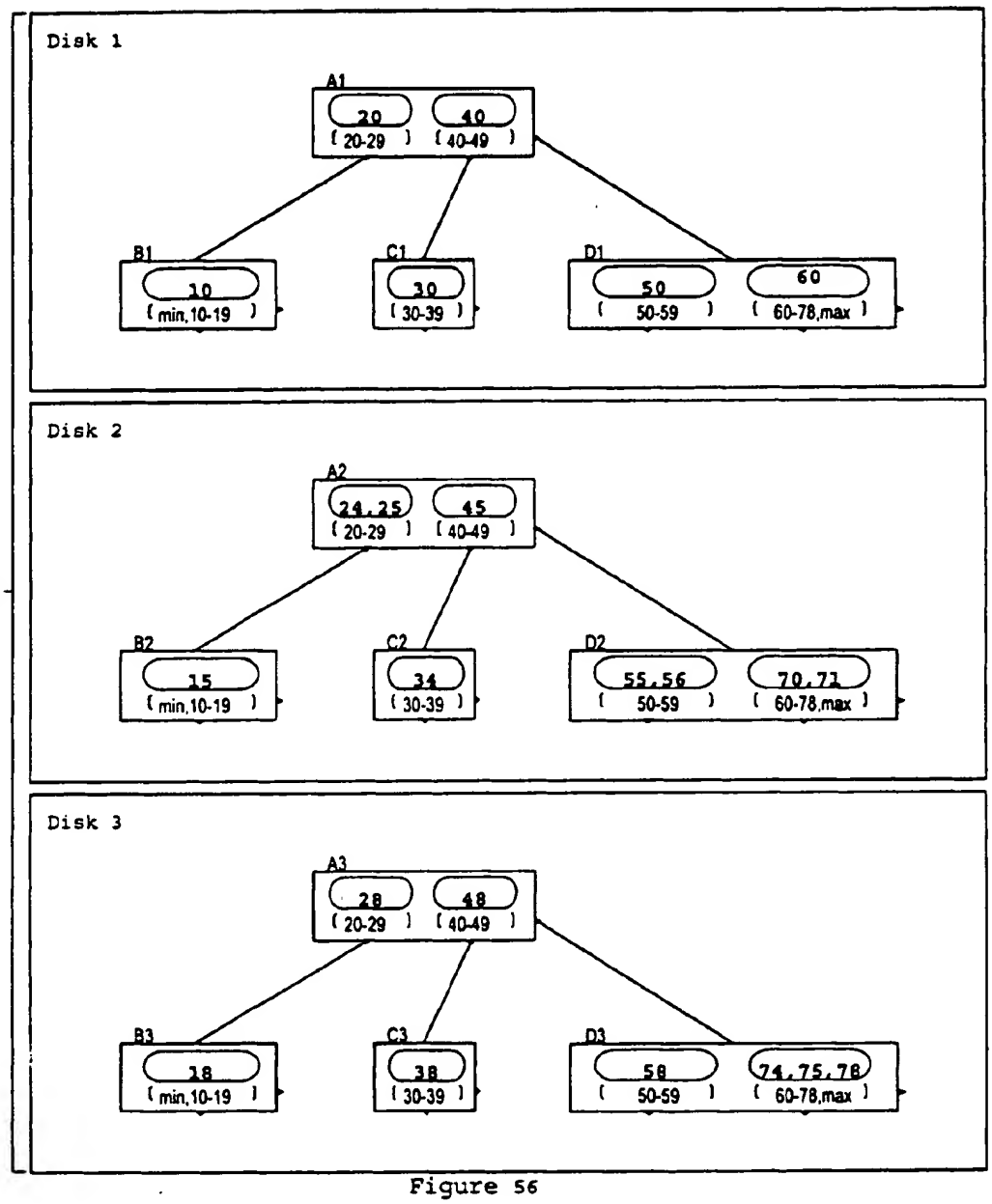



Fig 56

Figure 56

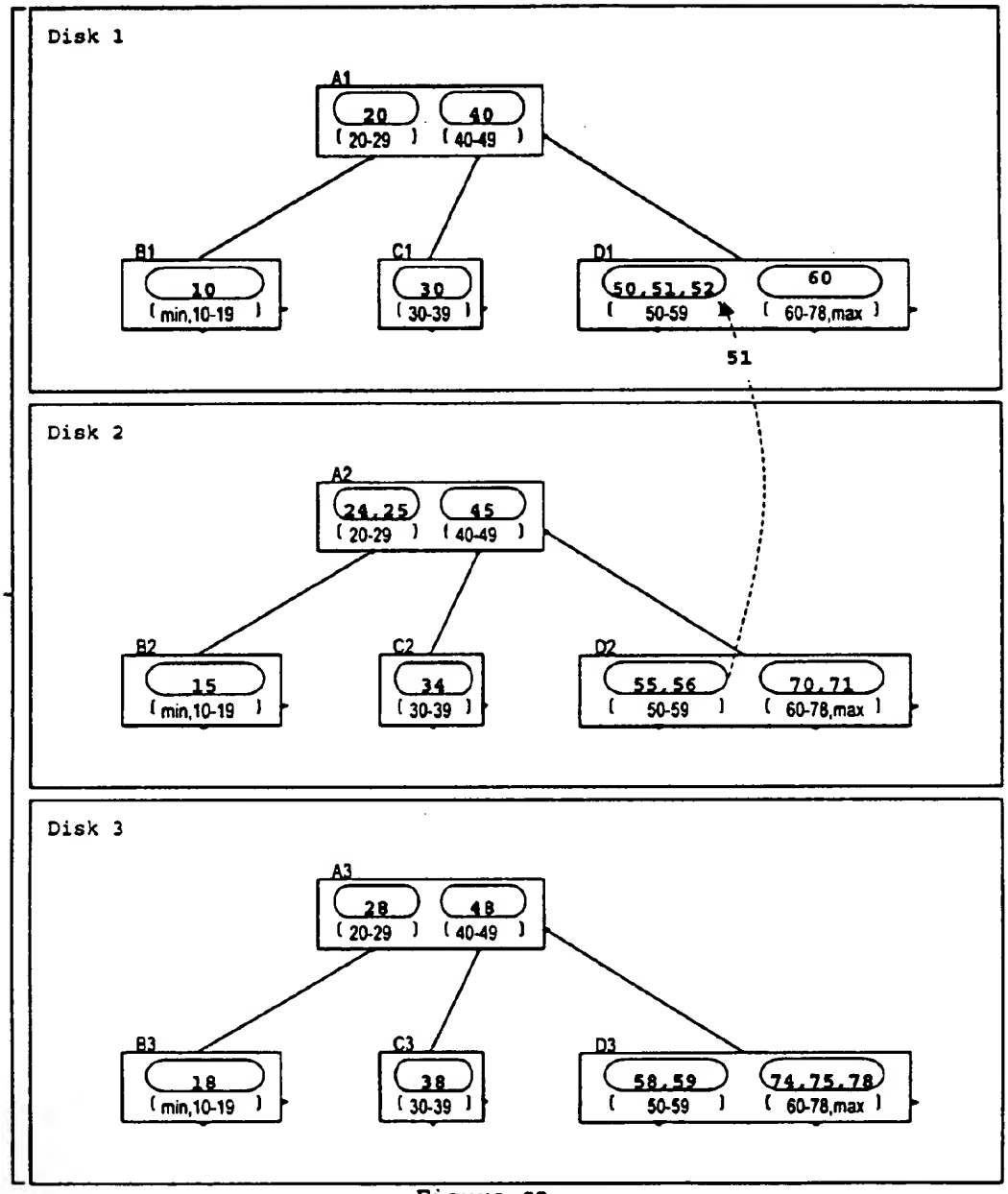



Fig 57

Figure 57

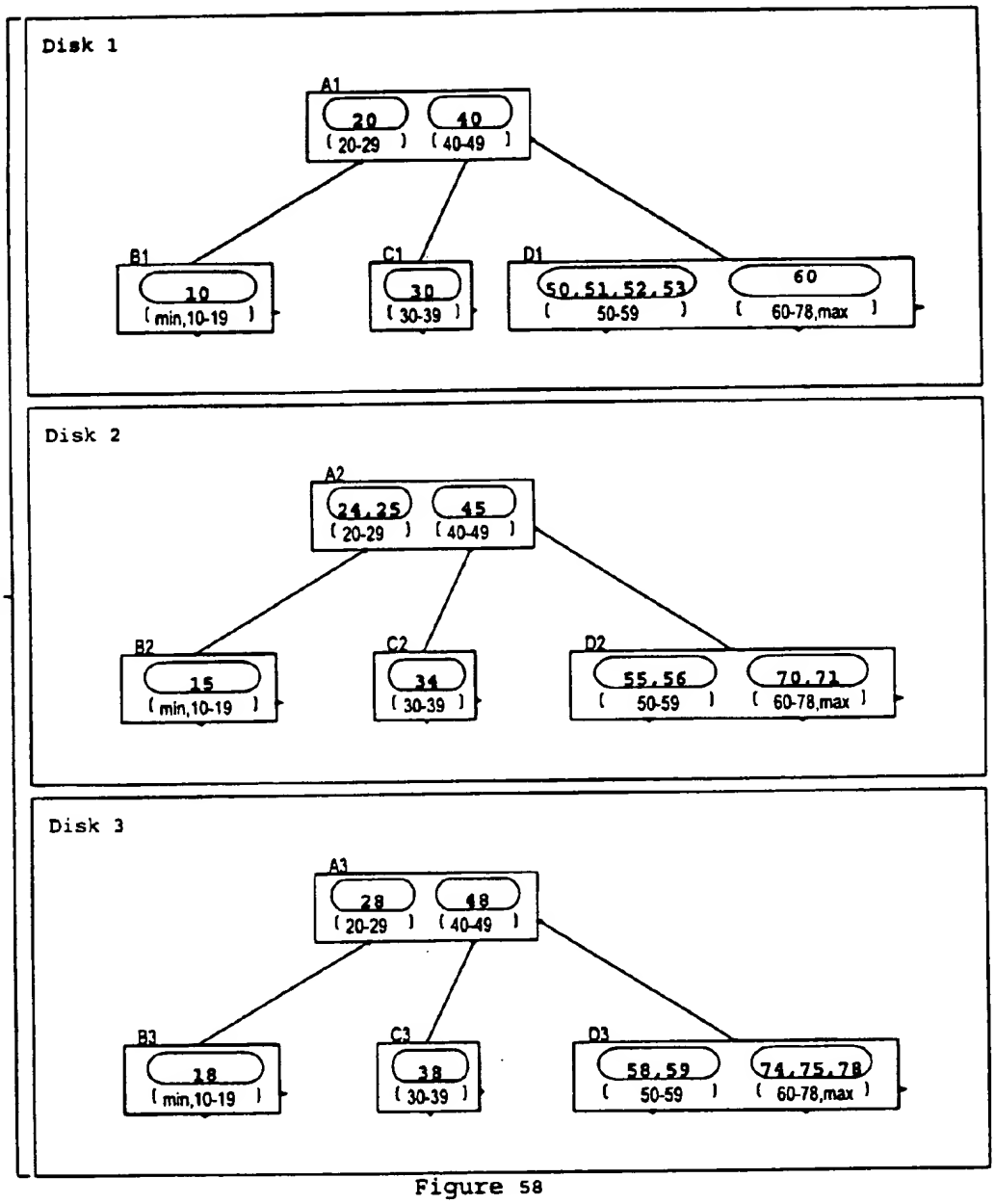



Fig 58

Figure 58

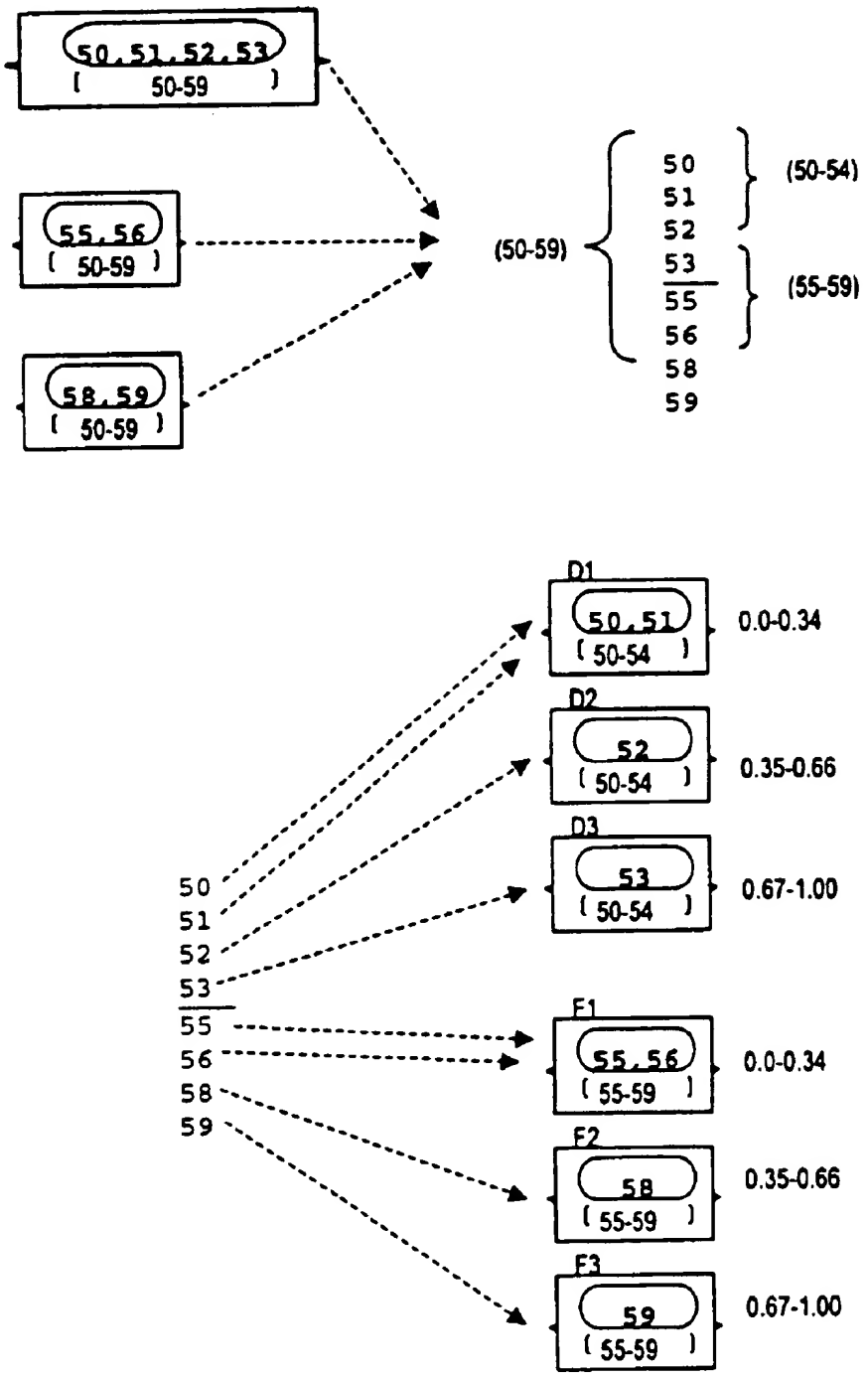

50,51,52,53
[ 50-59 ]
55,56
[ 50-59 ]
58,59
[ 50-59 ]
(50-59)
50
51
52
53
55
56
58
59
(50-54)
(55-59)
D1
50,51
[ 50-54 ]
0.0-0.34
D2
52
[ 50-54 ]
0.35-0.66
D3
53
[ 50-54 ]
0.67-1.00
50
51
52
53
55
56
58
59
F1
55,56
[ 55-59 ]
0.0-0.34
F2
58
[ 55-59 ]
0.35-0.66
F3
59
[ 55-59 ]
0.67-1.00

Figure 59

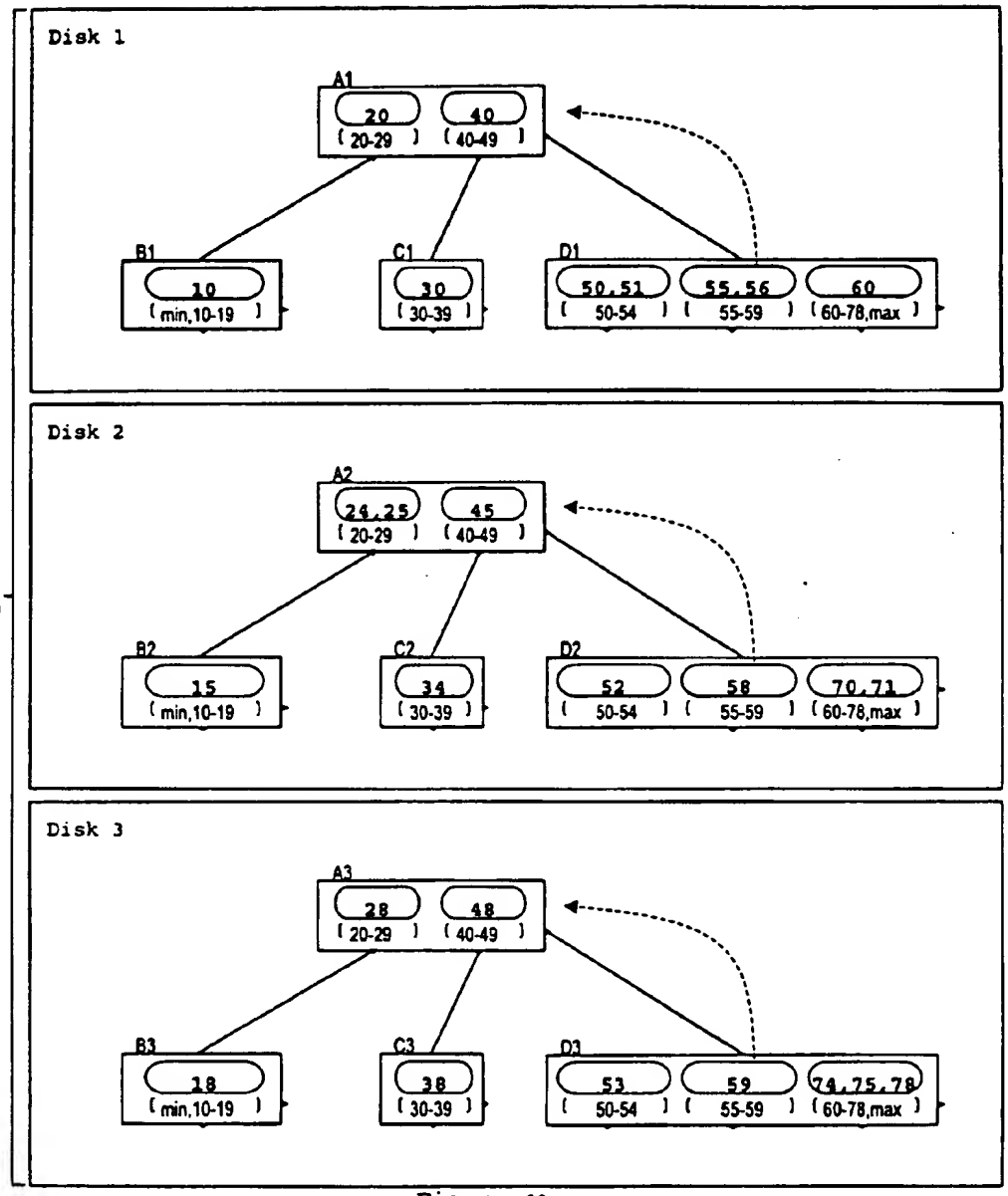



Fig 60

Figure 60

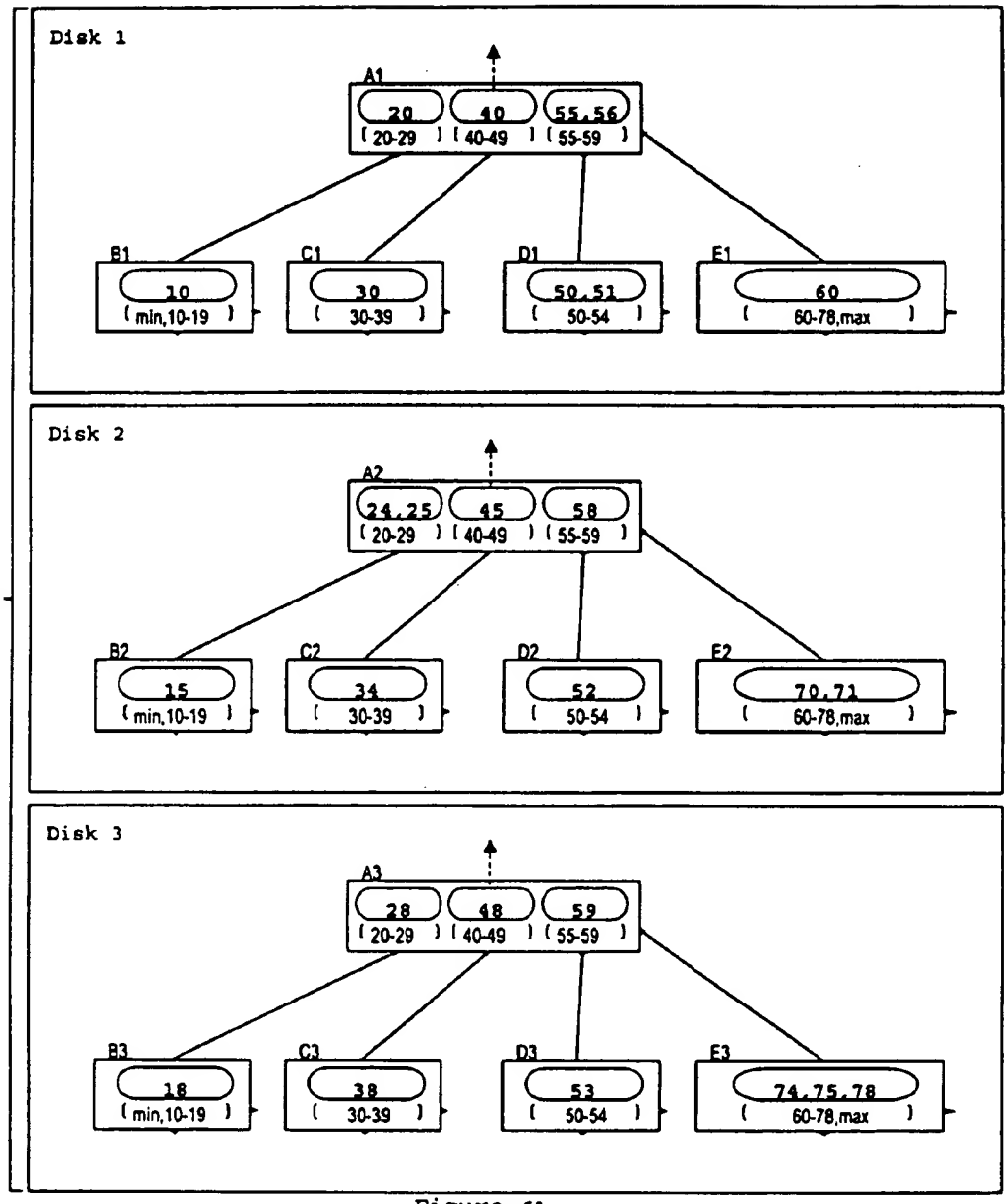



Fig 61

Figure 61

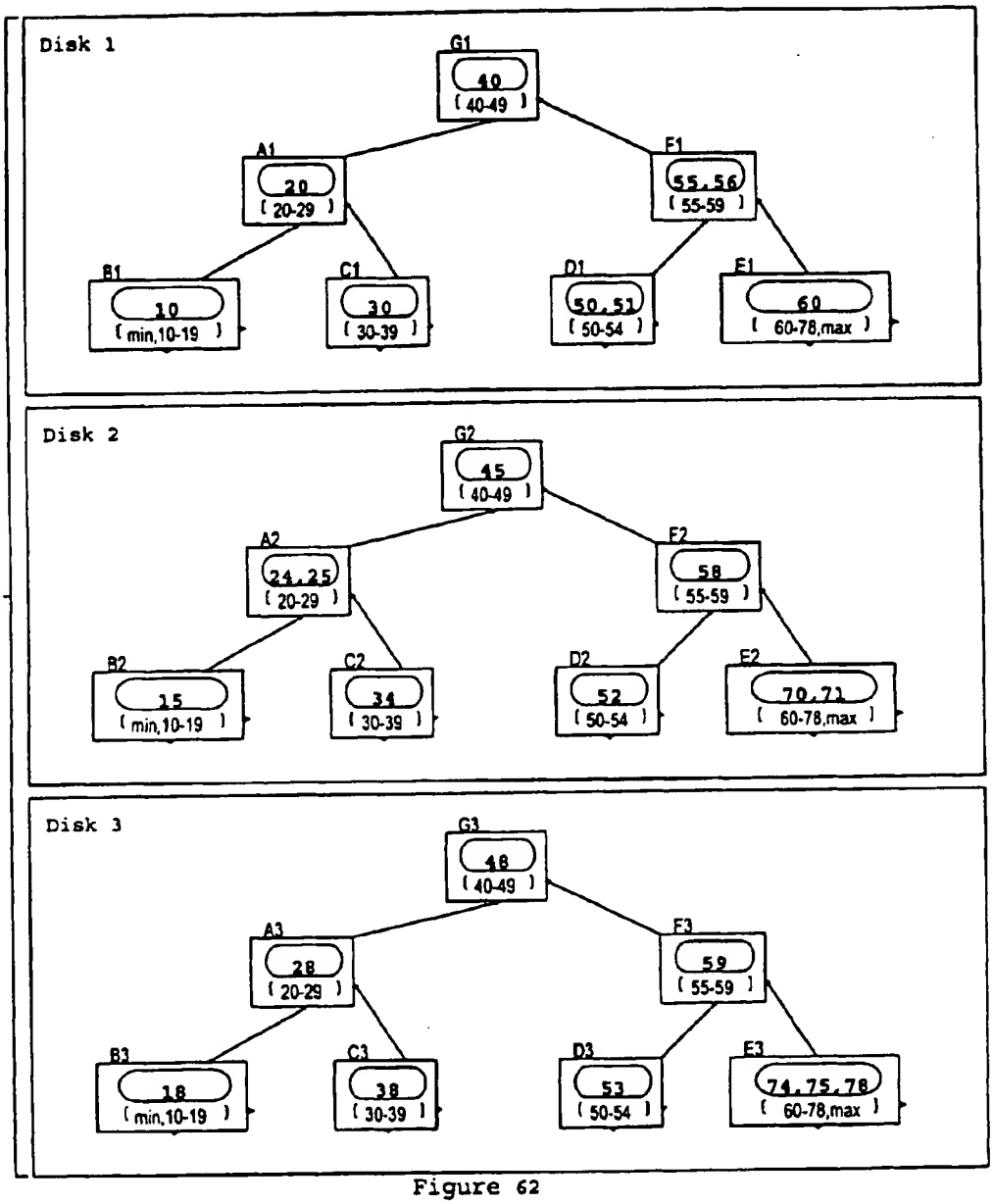



Fig 62

Figure 62

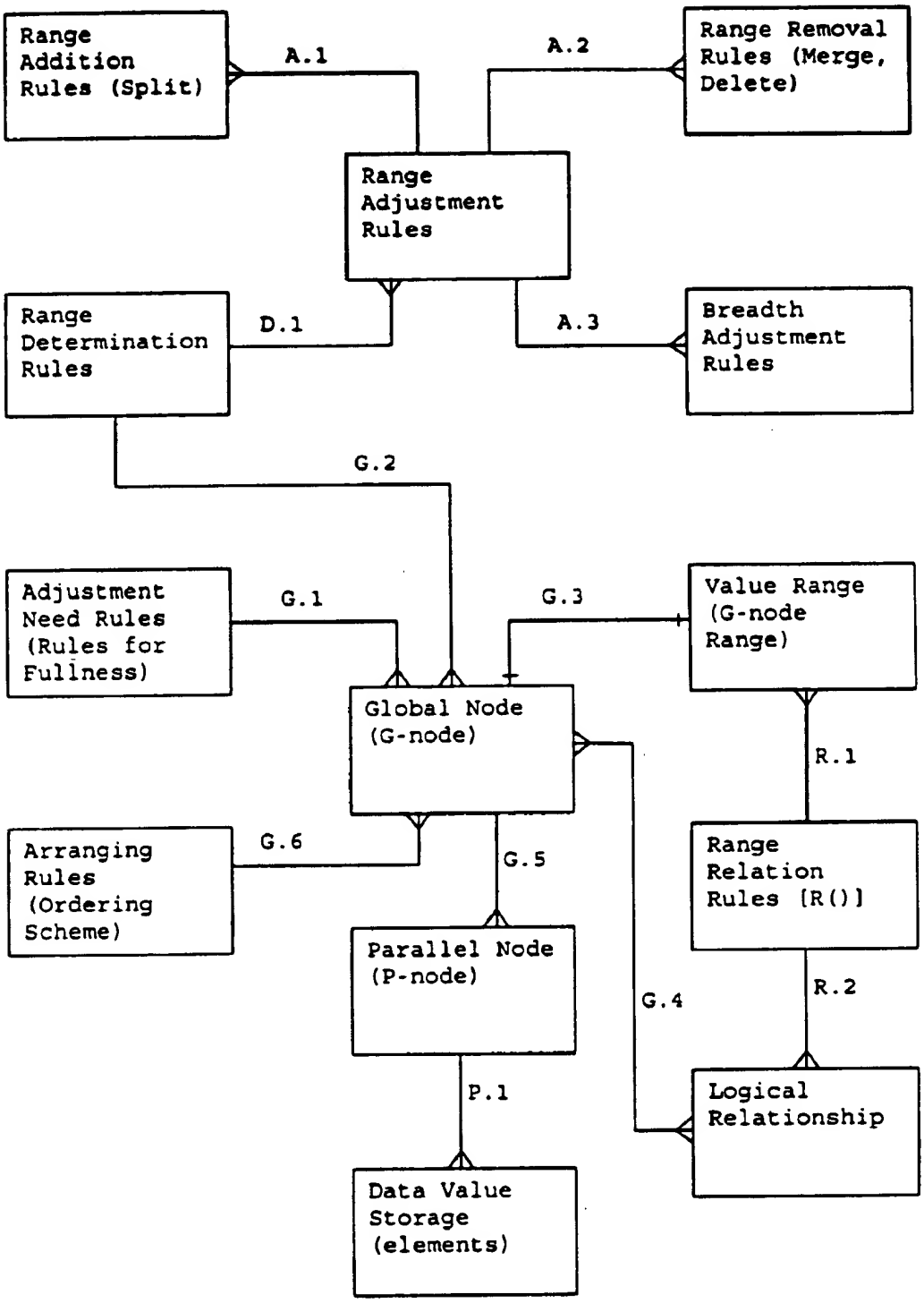

Range Addition Rules (Split)
A.1
A.2
Range Removal Rules (Merge, Delete)
Range Adjustment Rules
Range Determination Rules
D.1
A.3
Breadth Adjustment Rules
G.2
Adjustment Need Rules (Rules for Fullness)
G.1
G.3
Value Range (G-node Range)
Global Node (G-node)
R.1
Arranging Rules (Ordering Scheme)
G.6
G.5
Range Relation Rules [R()]
Parallel Node (P-node)
G.4
R.2
P.1
Logical Relationship
Data Value Storage (elements)

Figure 63

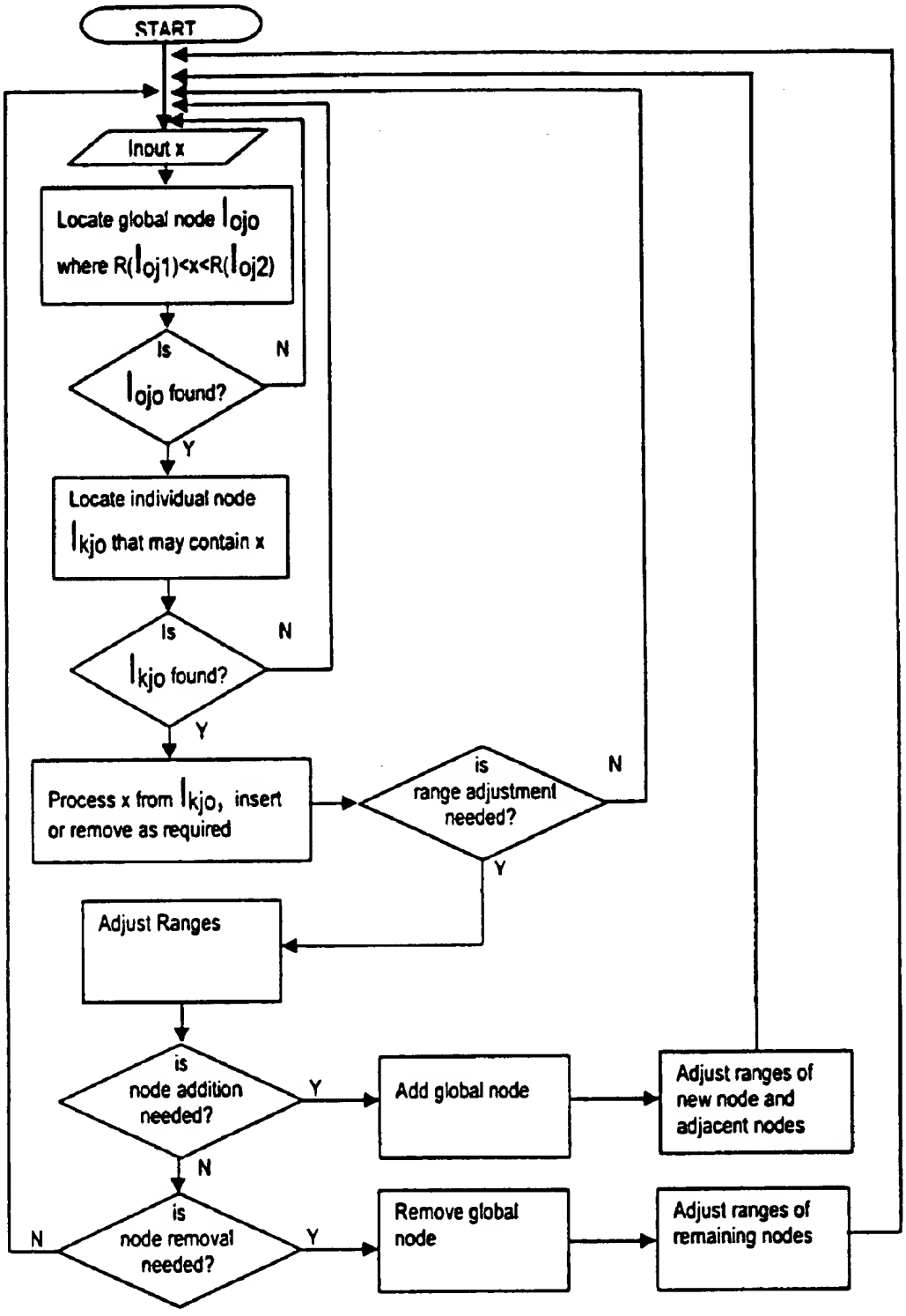

START
Inout x
Locate global node $l_{ojo}$ where $R(l_{oj1})<x<R(l_{oj2})$
Is $l_{ojo}$ found?
N
Y
Locate individual node $l_{kjo}$ that may contain x
Is $l_{kjo}$ found?
N
Y
Process x from $l_{kjo}$, insert or remove as required
is range adjustment needed?
N
Y
Adjust Ranges
is node addition needed?
Y
Add global node
Adjust ranges of new node and adjacent nodes
N
is node removal needed?
N
Y
Remove global node
Adjust ranges of remaining nodes

Figure 64

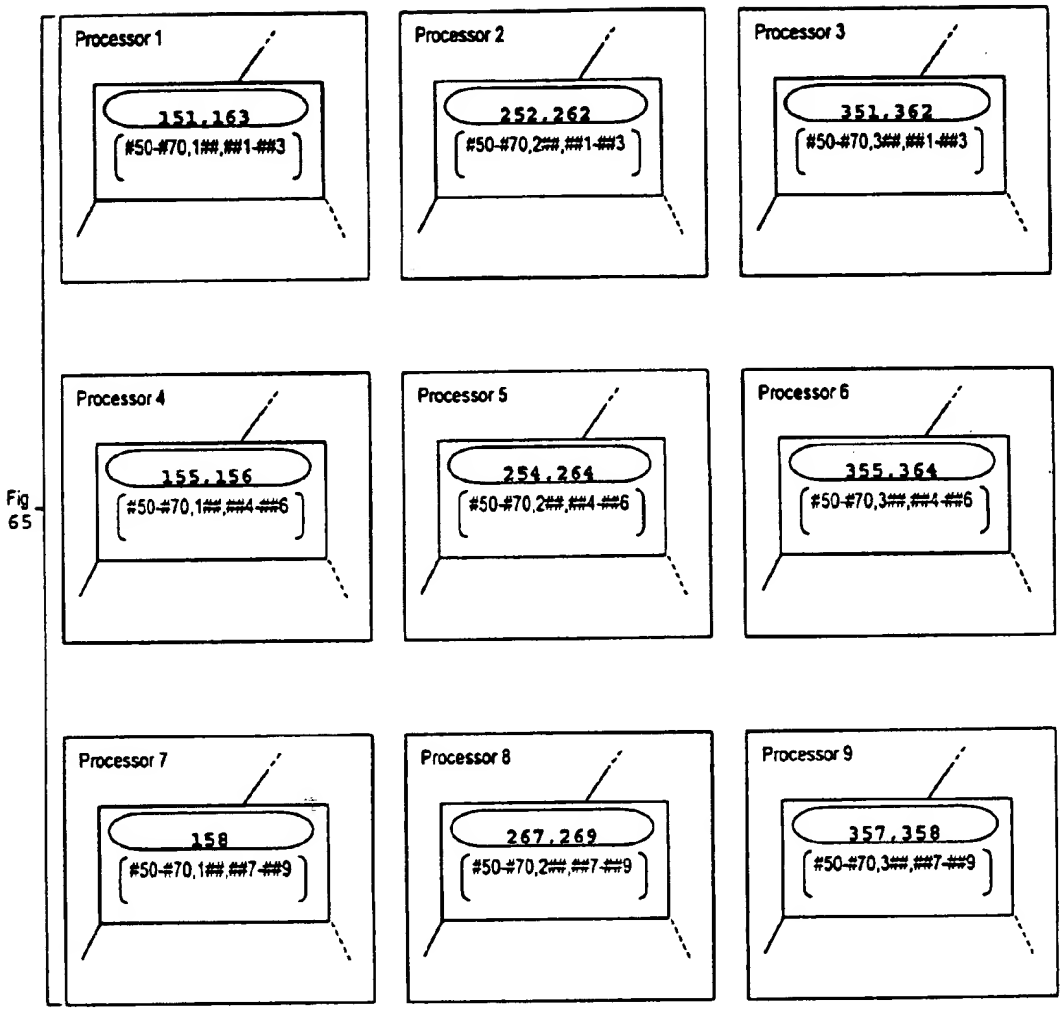



Fig 65

Figure 65

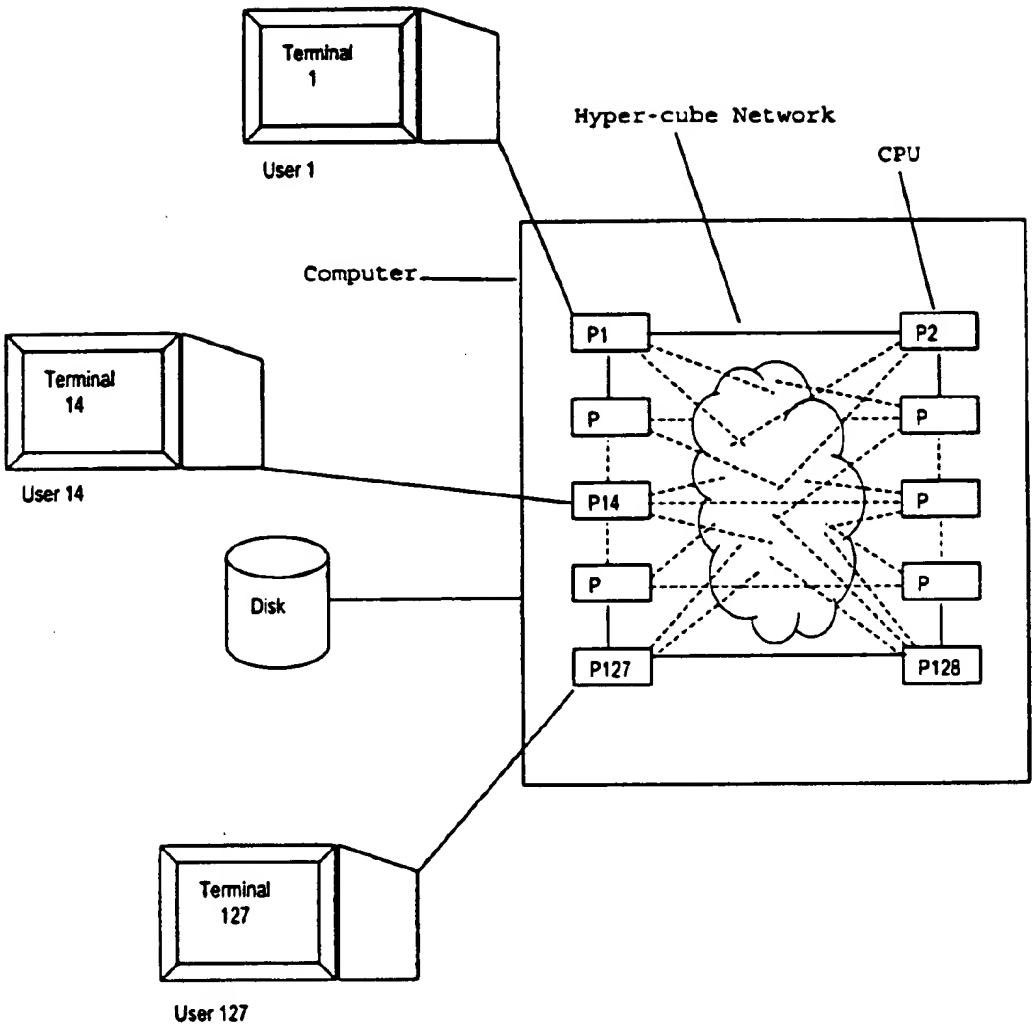

Terminal
1
User 1
Hyper-cube Network
CPU
Computer
P1
P2
P
P
Terminal
14
User 14
P14
P
P
Disk
P
P
P127
P128
Terminal
127
User 127

Figure 66

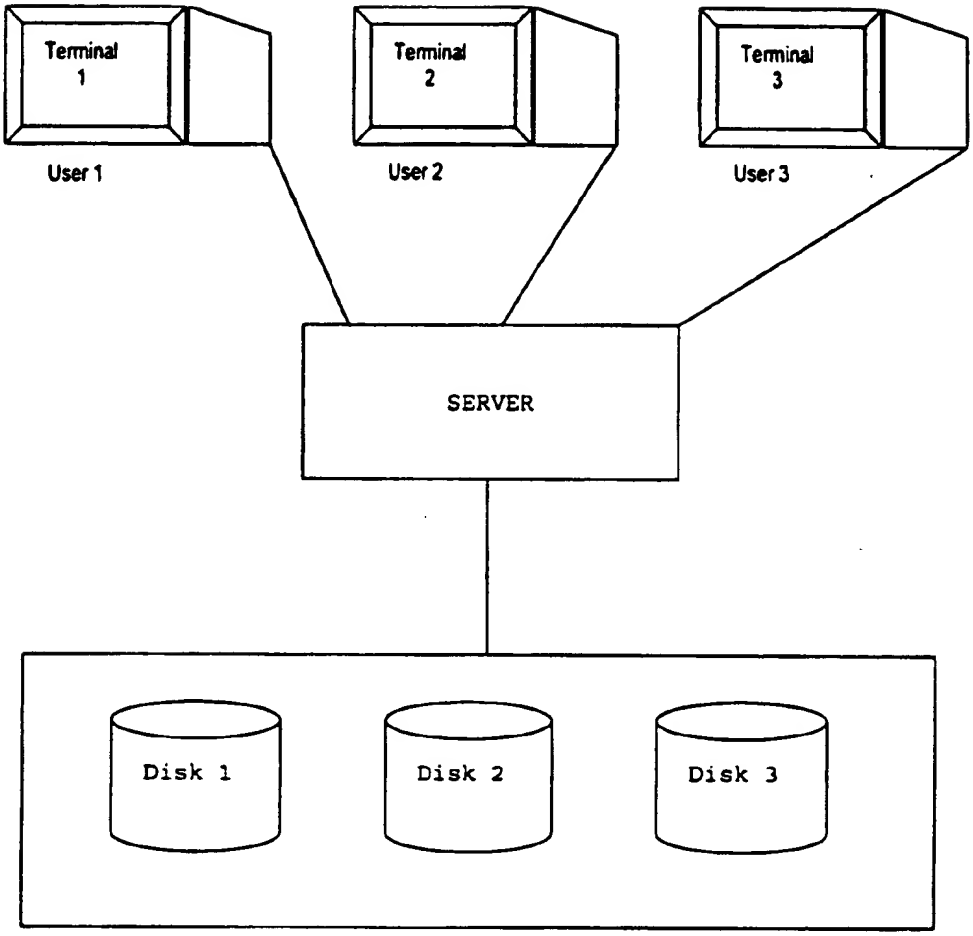

Terminal 1
Terminal 2
Terminal 3
User 1
User 2
User 3
SERVER
Disk 1
Disk 2
Disk 3

Figure 67

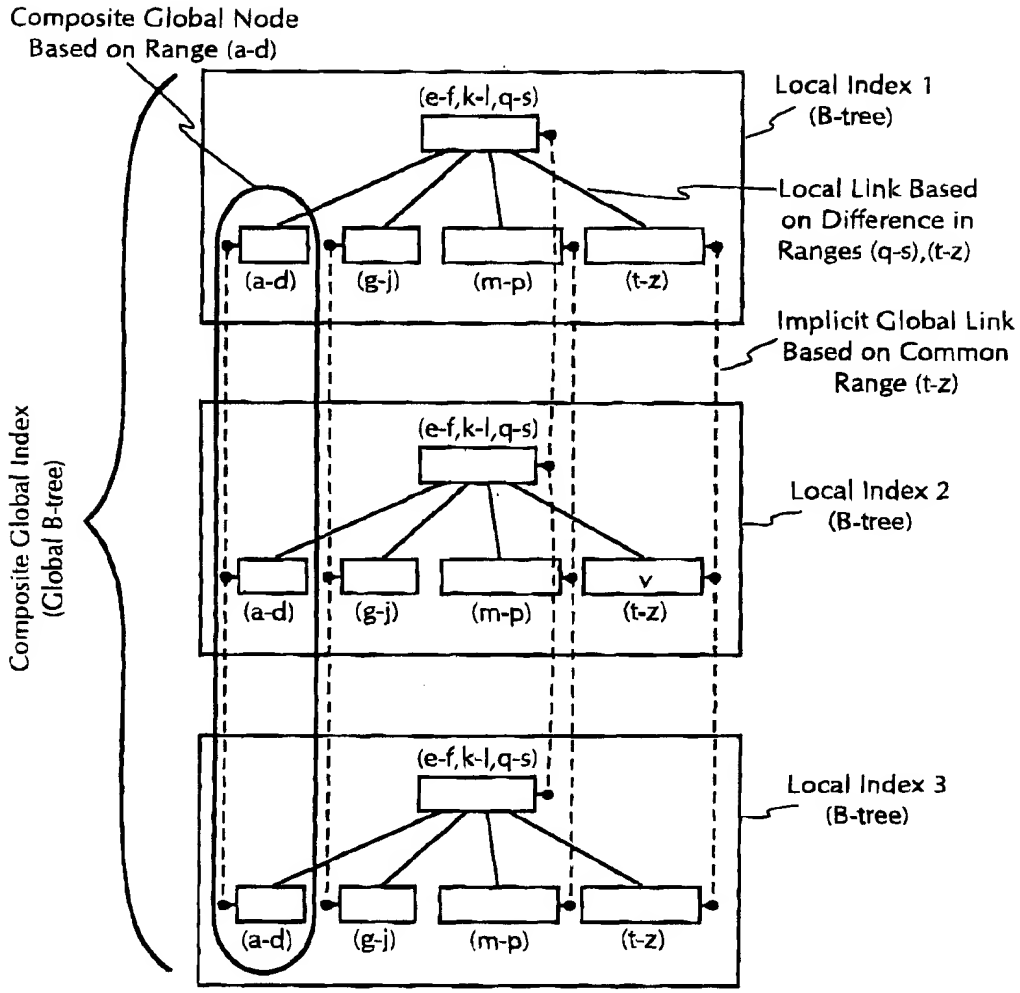

Composite Global Node
Based on Range (a-d)
Composite Global Index
(Global B-tree)
(e-f,k-l,q-s)
Local Index 1
(B-tree)
Local Link Based
on Difference in
Ranges (q-s),(t-z)
(a-d)
(g-j)
(m-p)
(t-z)
Implicit Global Link
Based on Common
Range (t-z)
(e-f,k-l,q-s)
Local Index 2
(B-tree)
(a-d)
(g-j)
(m-p)
v
(t-z)
(e-f,k-l,q-s)
Local Index 3
(B-tree)
(a-d)
(g-j)
(m-p)
(t-z)

FIG. 68

# METHOD FOR CREATING AND USING PARALLEL DATA STRUCTURES

## CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims priority under 35 U.S.C § 119(e) to the following related provisional applications: Ser. No. 60/023,340, filed Jul. 25, 1996 and Ser. No. 60/022,616, filed Jul. 26, 1996.

## BACKGROUND OF PROBLEM AND SOLUTION

In recent years, the need for more computational power and speed in computer systems has lead to the use of multiple processors to perform computational tasks. The processors work cooperatively, sharing resources and distributing work amongst themselves by sending data over communication lines or through shared memory. This practice of utilizing multi-processors to accomplish a single task is known as parallel processing or distributed processing. Although the terms parallel and distributed may describe distinct forms of multi-processing, they are in essence synonymous. The problems described and solved by the present invention apply equally to parallel and distributed processing. In addition, these problems/solutions apply to any component or aspect of a computer system to which work may be distributed amongst multiple components, even in non-parallel systems: one such "non-parallel" application described herein is the use of the present invention to manage dynamic access storage devices (DASD) such as disk drives [see section *Rules for Fullness and Ordering Scheme for B-trees Stored on Disk*].

Dividing work amongst multi-processors in such a way that the work is divided evenly and performed in an efficient manner is the goal of parallel/distributed processing, and dividing work amongst multiple system components equally and efficiently is also desirable in sequential (single-processor) systems. Many well-known sequential methods, systems or processes exist to efficiently perform computational tasks (sorting, merging, etc.). Their parallel counterparts have yet to be invented. Some new parallel methods are parallelized versions of existing sequential methods, for example, the parallel recursive merge-sort [see "Introduction to Parallel Methods" by Joseph JaJa, Addison-Wesley, 1992]. The invention described herein is a method of creating parallel data-structures. The preferred embodiment of the present invention parallelizes single-processor, ordered list methods to efficiently distribute the work and storage for ordered list maintenance amongst multiple processors, processing components and/or storage locations: this ordered list maintenance is carried out through adapted versions of single-processor data-structures expressed as graphs (B-trees, AVL trees, linked-lists, m-way trees, heaps, etc.).

## DISCUSSION OF PROBLEM AND PREFERRED EMBODIMENT

The goal in parallel processing is to utilize a number of processors (P) to increase the system's speed and power by a factor of P: optimally, a task requiring time T on a single-processor can be accomplished in time T/P on P processors. The problem is the even distribution of work amongst the P processors. Many new methods have arisen from the field to efficiently distribute the work for standard computational tasks (e.g. sorting, merging, etc.). One standard task is the maintenance of ordered lists of data: many methods exist for single-processor systems to accomplish



ordered list maintenance; the problem described and solved herein is the efficient distribution of work amongst multi-processors to accomplish efficient ordered list maintenance. (The term "ordered list" includes many data-structures: sorted lists, heaps, stacks, trees, etc.).

In general (regardless of the type of system keeping the lists), the maintenance of ordered lists consists of two basic operations: Insert() and Remove() (Search()/Find() is implied.). Insertion into the lists requires that an element of data be added to the list and that its position within the list be defined. Assuming the ordered list {5,12,46,67,80,99}, the Insertion (Insert(x)) of the numeric element **35** (Insert (35)) results in the list {5,12,35,46,67,80,99}. Removal (Remove(x)) of an element can take several forms: removal by location, by value, by range of values, etc. Again, assuming the list {5,12,46,67,80,99}, Removal of the fourth (4th) element results in {5,12,46,80,99}, Removal of the value 12 results in {5,46,67,80,99}, Removal of the smallest element greater than 50 results in {5,12,46,80,99}. The Remove operation is considered to return the value of the removed element for use, if present, or to return the information that a specific value is not contained in the list, if not present.

The problem presented and solved in the preferred embodiment is the parallelizing of the list maintenance described above. The essential functioning of the list remains the same in the parallel version of the data-structure. The Insert(x) and Remove(x) operations produce the same results. However, on a single-processor system these operations are performed by one processor which can only Insert or Remove one element at a time; on a multi-processor system with P processors, the parallel version of the method can Insert and/or Remove P elements at a time as described below.

Assuming a multi-processor system with 3 processors (P=3), and also assuming a list containing the elements {4,13,14,20,28,34,39,43,53,67,76,81} we have the following parallelized result: each processor keeps approximately one-third of the elements at any given time; each processor may Insert(x) into its own sub-list at any given time (possibly sending the element x to one of the other processors for Insertion into one of the other sub-lists); each processor may Remove(x) from its sub-list at any time and may request that other processors attempt to locate element x in their sub-lists if x is not present in the original processor's sub-list; any other processor finding x in its sub-list then sends x to the original processor.

The sub-lists are distributed in this example by cutting the list into equal thirds. This manner of distribution is for the purpose of a generalized example only. The Example given in this section is intended to introduce the reader to the problem in the most generalized manner possible; the Example here contains none of the specific details of the parallel method.

The Parallel List

Processor #1 (P1) keeps one-third of the elements:

Sub-list S1={4,13,14,20}

Processor #2 (P2) keeps one-third of the elements:

Sub-list S2={28,34,39,43}

Processor #3 (P3) keeps one-third of the elements:

Sub-list S3={53,67,76,81}

Insertion into the Parallel List

The elements **72**, **22**, and **12** are to be Inserted. All three processors simultaneously perform the Insertion giving the results:

Processor #1 (P1): Insert(72)—sends element **72** to P3, receives element **12** from P3

Sub-list S1={4,12,13,14,20}

Processor #2 (P2): Insert(22)—inserts element **22** directly into its sub-list (S2)

Sub-list S2={22,28,34,39,43}

Processor #3 (P3): Insert(12)—sends element **12** to P1, receives element **72** from P1

Sub-list S3={53,67,72,76,81}

Removal from the Parallel List (List Contains Elements from Insertion Above)

The values 37, 28 and 13 are to be found and Removed. All three processors simultaneously perform the Removal, giving the results:

Processor #1 (P1): Remove(37)—requests another processor to find 37 and receives reply from P2 that the element **37** is not present, receives request for element **13** from P3 and Removes **13** from the list.

Sub-list S1={4,12,14,20}

Processor #2 (P2): Remove(28)—removes **28** directly from the list, replies "37 not present" to P1

Sub-list S2={22,34,39,43}

Processor #3 (P3): Remove(13)—requests another processor to find 13, receives **13** from P1

Sub-list S3={53,67,72,76,81}

It must be stressed that the example above is a generalized example intended to explain the basic logical functionality of the problem. The precise details and organization of parallelized lists are described in subsequent sections.

The essential functioning of an ordered list is described above; however, many different forms of lists are used on modern systems, and many different types of data may be stored. Efficient methods/data-structures are used to maintain such lists on single-processor systems: heaps, binary trees, AVL trees, B-trees, etc which are well known in the art. (For descriptions of such methods/data-structures see "File Structures Using Pascal" by Nancy Miller, The Benjamin/Cummings Publishing Co., Inc. (1987)). The methods used on modern systems were designed to function on single-processor systems efficiently. This efficiency is expressed by asymptotical time-complexity functions. The functions are generally expressed in terms of n in the form O(f(n)) [e.g. $O(\log_2 n)$ or $O(n^2)$]. For the problem to be truly solved, a parallel version of a list maintenance method must distribute the work amongst the P processors efficiently so that the time-complexity approaches optimum improvement (speedup). Perfect speedup for a given parallelized method would be O(f(n)/P).

## SUMMARY

The present invention is a means to create parallel data-structures and associated maintenance programs. The data-structures and programs may take a variety of forms, all using the same essential steps and components. The parallel data-structures distribute a given data set to system components by grouping the data set according to ranges. These ranges are sub-divided for distribution into parallel form. A given data value is located by its placement within an appropriate range; the ranges are located by their relationships to each other and the data set as a whole; thus, as the ranges are related to each other, the order of the data set is maintained and access may be gained to the data set by range, and as the data values are related to the ranges, the data values themselves may be maintained as well.

In order for a data set to change, the values or the relationships between the values must change. The present invention allows this change by altering the ranges or the relationships between the ranges and thereby altering the values or relationships between values. Altering a range may alter the sub-set of data contained by the range, and this range alteration may then be used to re-distribute data values and maintain appropriate sizes and locations for the data sub-sets. The maintenance of the ranges, sub-sets and data value distribution within the sub-sets offers a wide variety of possible over-all distributions of data sets and methods of maintaining order. Some of these distributions and methods are parallel forms of serial data-structures.

The present invention offers many advantages including: a flexible means to create a wide variety of parallel data-structures rather than simply defining a single instance of a particular parallel data-structure; flexible methods of distributing data within a structure for efficiency; the ability to create parallel versions of serial data-structures that maintain the essential efficiency and express the essential form of the serial data structures without significant alteration of the principles or methods that underlie the serial data-structures.

## OBJECTS AND ADVANTAGES

One object of the method of creating data-structures is to distribute work and storage to multiple system components. The method can accomplish the distribution of work by allowing simultaneous access to multiple parallel nodes, graphs or indexes by multiple processing elements in a flexible manner. It can accomplish the distribution of storage by distributing multiple parallel nodes to multiple storage locations.

Another object is to provide the ability to distribute data more evenly. A data set with a skewed distribution may be more evenly distributed by breaking the data into sub-sets. Each sub-set may be distributed evenly while all of the sub-sets taken together still express the original distribution of the data set.

An advantage of the method when used to transform serial data-structures into parallel form is that the original structure of the serial algorithm can be expressed without altering the essence of the algorithm.

Another advantage is the wide range of possible structures created. Many serial data-structures may be adapted using the same principles as well as many new parallel data-structures created.

Another advantage is the use of various components of the method to refine the functioning, data distribution, work distribution and efficiency of the data-structures and associated maintenance programs through the characteristics of the rules that support the various components. For only one example, see the *Rules for Fullness and Ordering Scheme for B-trees Stored on Disk* section contained herein.

Still other objects and advantages will become apparent through a consideration of the other descriptions of the invention contained herein.

## BRIEF DESCRIPTION OF FIGURES

FIG. **1** shows serial b-tree.

FIG. **2** shows parallel b-tree on two processors with indication of G-node and P-nodes for preferred embodiment.

FIG. **3** shows parallel b-tree of FIG. **2** after removal of one G-node.

FIG. **4** shows serial AVL tree of Example 1 for preferred embodiment.

FIG. **5** AVL tree of FIG. **4** after addition of element.

FIG. **6** AVL tree of FIG. **4** after rotation.

FIG. **7** AVL tree of FIG. **4** after another addition.

FIG. 8 AVL tree of FIG. 4 after another addition.

FIG. 9 AVL tree of FIG. 4 after rotation.

FIG. 10 AVL tree of FIG. 4 after removal of element.

FIG. 11 parallel AVL tree of Example 1 for preferred embodiment, comprising 3 separate trees stored on 3 processors.

FIG. 12 AVL tree of FIG. 11 after addition of element.

FIG. 13 AVL tree of FIG. 11 after another addition.

FIG. 14 shows redistribution of elements to maintain Ordering Scheme.

FIG. 15 shows range split and redistribution of elements resulting in creation of new G-node (G-node Split) for Examples 1 and 2 of preferred embodiment.

FIG. 16 AVL tree of FIG. 11 after insertion of G-node.

FIG. 17 AVL tree of FIG. 11 after rotation.

FIG. 18 AVL tree of FIG. 11 after another addition of elements.

FIG. 19 shows redistribution of elements to maintain Ordering Scheme.

FIG. 20 shows range split and redistribution of elements resulting in creation of new G-node (G-node Split) for Examples 1 and 2 of preferred embodiment.

FIG. 21 AVL tree of FIG. 11 after insertion of G-node.

FIG. 22 AVL tree of FIG. 11 after another addition of elements.

FIG. 23 shows redistribution of elements to maintain Ordering Scheme.

FIG. 24 shows range split and redistribution of elements resulting in creation of new G-node (G-node Split) for Examples 1 and 2 of preferred embodiment.

FIG. 25 AVL tree of FIG. 11 after insertion of G-node.

FIG. 26 AVL tree of FIG. 11 after rotation.

FIG. 27 shows removal of elements from tree of FIG. 11.

FIG. 28 is shows another removal of elements from tree of FIG. 11.

FIG. 29 shows result of G-node removal from tree of FIG. 11.

FIG. 30 shows serial B-tree of Example 2 for preferred embodiment, comprising 3 separate trees stored on 3 processors.

FIG. 31 B-tree of FIG. 30 after addition of element.

FIG. 32 B-tree of FIG. 30 after b-tree node split.

FIG. 33 B-tree of FIG. 30 after additional b-tree node split.

FIG. 34 B-tree of FIG. 30 after another addition.

FIG. 35 B-tree of FIG. 30 after another addition.

FIG. 36 B-tree of FIG. 30 after b-tree node split.

FIG. 37 B-tree of FIG. 30 after removal of element and b-tree node merge.

FIG. 38 parallel B-tree of Example 2 for preferred embodiment.

FIG. 39 B-tree of FIG. 38 after addition of element.

FIG. 40 B-tree of FIG. 38 after another addition.

FIG. 41 shows redistribution of elements to maintain Ordering Scheme.

FIG. 42 B-tree of FIG. 38 after insertion of G-node.

FIG. 43 B-tree of FIG. 38 after b-tree node split.

FIG. 44 B-tree of FIG. 38 after additional b-tree node split.

FIG. 45 B-tree of FIG. 38 after another addition of elements.

FIG. 46 shows redistribution of elements to maintain Ordering Scheme.

FIG. 47 B-tree of FIG. 38 after insertion of G-node.

FIG. 48 B-tree of FIG. 38 after another addition of elements.

FIG. 49 shows redistribution of elements to maintain Ordering Scheme.

FIG. 50 B-tree of FIG. 38 after insertion of G-node.

FIG. 51 B-tree of FIG. 38 after b-tree node split.

FIG. 52 shows removal of elements from tree of FIG. 38.

FIG. 53 shows another removal of element from tree of FIG. 38.

FIG. 54 shows result of G-node removal from tree of FIG. 38.

FIG. 55 parallel B-tree stored on three disks for Example of B-trees Stored on Disk section.

FIG. 56 B-tree of FIG. 55 after element addition.

FIG. 57 shows redistribution of elements to maintain Ordering Scheme.

FIG. 58 B-tree of FIG. 55 after another addition of elements.

FIG. 59 shows range split and redistribution of elements resulting in creation of new G-node (G-node Split) for B-tree of FIG. 55.

FIG. 60 B-tree of FIG. 55 after insertion of G-node.

FIG. 61 B-tree of FIG. 55 after b-tree node split.

FIG. 62 B-tree of FIG. 55 after additional b-tree node split.

FIG. 63 data model for a preferred instance of present invention.

FIG. 64 flow chart for a preferred instance of present invention.

FIG. 65 shows nine P-nodes related by complex G-node Range.

FIG. 66 diagram of hypercube network with terminals and disk storage for Example of Application 1.

FIG. 67 diagram of distributed network showing three client terminals, one server and three disk-packs for Example of Application 2.

FIG. 68 is a block diagram illustrating the principles of the invention.

## PREFERRED EMBODIMENT

(3)

Introduction

The preferred embodiment of present invention relates to a process of creating parallel data-structures which adapts sequential data-structures and their associated processing programs for use in parallel or distributed environments. The invention achieves this by creating parallel data-structures that are identical in form and function to the sequential data-structures in a parallel environment. The adapted parallel data-structures and methods can be used in the same way as their sequential counterparts but in a parallel environment. The sequential data-structures which are adapted must have configurations determined by the orderable qualities of the data contained in the data-structures. The sequential data-structures and their associated maintenance programs generally have three functions in common: Find(), Insert() and Remove() functions.

FIG. 58 depicting applicant's parallel indexing method. FIG. 68 depicts composite Global Index comprising three local indexes; one of the five composite global nodes is circled and labeled (a–d); figure shows common structure

and access methods for all indexes, common ranges on all indexes, local links based on range difference, global links based on range commonality, and storage of one value "v" on index 2 in Range (t–z). A query on the value "V" can originate on any index 1, 2 or 3 equivalently. Assuming the query starts on index 1, the request may be shunted immediately to either index 2 or 3 if the processor for index 1 is busy: indexes 2 and 3 could also pass control to any other local index. The query travels down the rightmost local link on the controlling processor, locating the range (t–z) as the range to hold v; in the preferred embodiment, index 2 is immediately calculated as the specific index to hold V within the range (t–z) by virtue of its being the center of range (t–z) If index 2 does not already have control, the query then traverses the rightmost global link to index 2 at range (t–z) and index 2 accesses the range and thereby the value within (v). This process produces at least three different paths to v chosen dynamically at query time; if the query started on local index 2, then it requires 2 accesses (1 at the root plus 1 at the local node (t–z); if it started on either local index 1 or 3, then it requires 3 accesses ( 2 for the local index plus 1 at index 2 ).

In the parallel data-structures, each processor may contain multiple elements at any position v within the structure. The number of elements contained at position v is determined by the Rule for Fullness and Ordering Scheme for the given parallel data-structure. The simplest Rule and Scheme allow zero (0) through two (2) elements per processor to be contained at any position v. Such a simple Rule and Scheme are assumed for the introduction and any other section of this application unless otherwise stated.

In the Insert() function, for sequential data-structures, the insertion of an element y [Insert(y)] results in the placement of the element y in the sequential data-structure at some position V. The position v is determined the element y's orderable or ordinal relationship to the other elements and the positions of the other elements in the data-structure. The position v is determined by the "rule for insert" for the given sequential data-structure. Position v is also determined by the rule for insert in the parallel data-structure: each processing element creates a configuration for the data-structure identical to the configurations at all other processing elements.

Using the Rule for Fullness and Ordering Scheme mentioned above for parallel data-structures, each processor may contain as many as two elements $y_1$, and $y_2$ at position v. Consequently with P identical data-structures, one at each processor, there exist in total $1 \leqq n \leqq (2P-1)$ elements $y_{ij}$ $(1 \leqq i \leqq P)$ $(1 \leqq j \leqq 2)$ at all positions v, taken cumulatively. Any processor i $(1 \leqq i \leqq P)$ may insert any element y into the parallel data-structure, and the element y will be placed at position v in one of the data-structures held by one of the processors. Although this may result in different configurations for the sequential and parallel versions of the structures, the essential relationships between the data elements in the data-structures will remain the same for both versions of a given data-structure.

The Remove() function for sequential data-structures has one of two forms. A Remove() according to position finds an element in a given position in the data-structure and removes the element. A Remove() according to value, searches the data-structure for a given value of y and removes the element. In both cases, the data-structure may be re-ordered to compensate for the absence of the removed element. In an adapted parallel version of a given data-structure, any or all of the processors may execute a Remove(y) function appropriate to the sequential data-structure with the same result.

For parallel inserts and removes, a given processor simply locates the position v in the data-structure either by position or value. Multiple processors may then cooperate to search for a desired data element y within position v. The processors search through the $1 \leqq n \leqq (2P-1)$ elements at the positions v at all processors i $(1 \leqq i \leqq P)$ in parallel. If the removal of an element y at position v leaves position v sufficiently empty, then each processor re-orders its data-structure according to the missing position v corresponding to the element y in the same way that the sequential method would. If the addition of an element y requires more nodes to contain the larger data set, then each processor re-orders its data-structure according to an additional position w corresponding to the element y in the same way that the sequential method would.

Preferred Embodiment—Uses of Data-Structures

The uses of the adapted parallel versions of the data-structures and maintenance programs are the same as the uses of their sequential counter-parts, only in a parallel environment. The speedup of the parallelization brought about by the present method is very efficient and justifies its design.

Preferred Embodiment

Definitions

Many terms must be defined to adequately describe the Process of Adaptation.

At Will (Implies Blind)—an activity that a processor may perform at any time regardless of the activities of other processors;

Blind (Blindly)—Activities performed by a processor or set of processors with no cooperation from other processors;

Cooperative (cooperatively)—activities performed by a set of processors that require communication and/or coordination between processors;

Data-structure—an organization or method of organization for data. Preferably, the data-structures are based (form or configuration and functioning) on orderable data; e.g. heaps, B-trees, binary search trees, etc.;

Defined G-node: See G-node

Element—a single data-value within a data-structure. Each element may be of any type (e.g. integer, real, char, string, enumerated, pointer, record, or other). The elements must all relate to each other in some orderable fashion;

Element Deletion—Removal of an element from a G-node;

Element Addition—Insertion of an element to a G-node;

Explicit G-node Range—see G-node Range

Global—all of the processors;

G-node (Global Node)—a set of P(number of processors) P-nodes. Each G-node contains $0<n<(xP)$ elements (x=Max number of elements in each P-node). In the preferred embodiment, each P-node in a G-node occupies the same position in each per-processor data-structure. Each P-node in a G-node contains the G-node Range of that G-node. The G-node functions in the parallel method in the same way an S-node functions in the corresponding sequential method. The G-node uses the G-node Range to relate to the other G-nodes. G-nodes are created simultaneously with the P-nodes which are contained in the G-node.

G-nodes have the following properties: each has a G-node Range; all the G-nodes in a parallel data-structure may become full or empty or partially empty; when a G-node becomes full, it is Split; when a G-node becomes sufficiently empty, it is deleted. The determination of when a G-node is full or sufficiently empty depends on the Rule for Fullness

for the G-node. Each G-node is composed of P sets of data elements within the G-node Range; each of the P sets may contain from 0 to X elements. A Defined G-node is a G-node with a fully defined G-node Range. An Undefined G-node has a G-node Range with one or more boundaries left open or undefined.

G-node removal—deletion of a G-node from the data-structure; this effectively removes old G-node Ranges from the data-structure;

G-node insertion—Addition of a G-node from the data-structure; this effectively adds new G-node Ranges to the data-structure;

G-node Range—The G-node Range is the range of values that the G-node may contain, in the preferred instance, a set of two values $R(G_{oio})=\{R(g_{oi1}), R(g_{oi2})\}$ that are the minimum and maximum values of the elements which may be contained in the G-node G. The G-node Range determines the proper placement of the G-node within the parallel data-structure and thereby determines the proper placement of an element or P-node within each per-processor data-structure;

The G-node is stored across multiple processors, but the G-node Range uses the same range for each component of the G-node on each processor. The Range is stored with the G-node. The Range may be stored either explicitly or implicitly: explicit storage of the G-node Range is the listing of the values that define the range; implicit storage would be the storage of one or more values from which a range could be calculated.

G-node Split—A G-node Split occurs when a G-node becomes full. The Splitting process divides all of the values contained in the G-node into two roughly equal sets X and Y with distinct ranges. One set X remains in the G-node, the other set Y is stored in a newly created G-node. The G-node Ranges of the two nodes are set according to the division of the sets X and Y. The G-node Split is a method of adding new G-nodes to the set of G-nodes comprising the parallel data-structure; by virtue of the G-node Range as the basis for this process it is also a range addition method.

Implicit G-node Range—see G-node Range

Link—representation and reference to the relationship of adjacent nodes;

MAXVAL—Maximum possible value (oo)

MINVAL—Minimum possible value (-oo)

Ordering Scheme—The manner in which data elements are arranged within a G-node. May be ascending, descending, partially or fully sorted, completely unordered in addition to many other arrangements. Different Schemes may be defined for different data-structures. Schemes may be defined to provide efficient access paths, efficient data distribution, proper placement of an element into an appropriate P-node within a G-node, or other provisions;

Ordinable—data that has the capacity to be ordered;

P—number of processors on a parallel machine or distributed network;

Parallel—processes or entities performed or existing on multiple processing or memory storage units, designed to perform or exist on multiple processors or system components, or having a structure that lends itself to similar distribution;

Parallel Data-structure or Global Data-structure—the data-structure that results from applying this process of adaptation to a sequential data-structure. A parallel or Global data-structure is composed of a set of P sequential data-structures each of which is composed of a set of P-nodes and incident links. The P-nodes and links form precisely the same configuration on each processor.

Partially Defined G-node: See G-node

Partially Undefined G-node: See G-node

Per-processor—sequential activities on a processor or on a set of processors: this term conceptually divides a parallel entity or process into its sequential parts and refers to each processor's activity separately;

P-node (Parallel Node) an adaptation of an S-node. A P-node contains 0 to n elements which fall into its G-node Range. In addition, a P-node relates to the other P-nodes in the data-structure not only by the value of the elements contained in the P-node but also by the P-node's G-node Range which is contained in each P-node and determined by the G-node to which the P-node belongs. Each P-node in a data-structure is part of a G-node. When converting an S-node into a P-node, extra links are not added for the extra elements. The rules for relationships between P-nodes on a processor are the same as the rules for relationships between the S-nodes of the sequential data-structure from which the parallel version was derived with respect to G-node Ranges. Except for P-nodes created at the very beginning of the process, P-nodes are generally created through the splitting of G-nodes;

Processor—a processing element or CPU with or without its own local memory in a parallel or distributed environment. The processors are all interconnected in the parallel machine or network by communication lines or by shared memory. Also used to refer to any system component to which work may be distributed.;

Range Relation Function—This function R() determines how G-node Ranges relate to each other (i.e. less than, greater than, equal to, subsets of, supersets of each other, etc.);

Range Determination Rules—these rules determine ranges for the data: in the preferred instance, the range is based on data placement (the number, value, distribution and/or positions of element values for splits); however, ranges may also be set to force a change in the data placement, or set according to other criteria;

(48)

Rule for Fullness—The rule by which the fullness or emptiness of a G-node is determined. Full G-nodes are split; empty G-nodes are removed. Different Rules may be defined for different data-structures. The goal in setting the rules for determining the fullness of G-nodes is to make the most efficient use of space, and processing time. The Rule for Fullness expresses and may be used to maintain: range or G-node fullness, emptiness, range breadth (narrowness or broadness), density and distribution of data values within data structures, etc.;

Rule for Insert (Also referred to as Rule for Remove and Rule for Positioning Nodes)—the ordinal or orderable relationships between the data elements contained in the nodes of a given data-structure; especially in the Insert() and Remove() functions of sequential programs and data-structures and the same functions (with respect to G-node Ranges) for their parallel counterparts;

Sequential or Serial—processes or entities performed or existing on a single processor or designed to perform or exist on a single processor;

S-node (Sequential node)—a single cell within a sequential data-structure that contains a single element. Each S-node relates to its adjacent nodes or to the rest of the data-structure according to the ordinable relationships between the element the node contains and the elements contained in the other nodes;

Preferred Embodiment

Symbols

Data-structures—Sets of nodes containing elements, and incident links.

Elements An element is a single piece of ordinable data. Elements are generally depicted in one of two ways: 1. An element may be thought of as having a constant value; such an element usually belongs to a set that contains members with single subscripts: ($w=\{w_1, W_2, W_3, \ldots W_n\}$); 2. An element may also be referenced by its position in a data-structure; these are usually referenced by the subscripted cell letters to which they belong: ($x=\{x_{111}, x_{112}, x_{121}, x_{122} \ldots\}$ Elements are also frequently depicted as their actual values, both in set and graphical form.

G-node Ranges : A G-node Range is considered consistent over the entire G-node, and therefore has the same kind of notation at each P-node. Because the actual values of the range may be explicit or implicit, the Range is indicated by a function reference [R(G-node)]. The parameter G-node is expressed in the manner appropriate to the given example. If the function receives an element parameter, it may then be used to compare the element to ranges to determine proper placement of the element. The function R() may be called by any processor and may use any other additional parameters needed to calculate the Range. In the preferred instance, the result is a minimum and maximum value allowable for the P-node and/or G-node: $R(A_{oio})=\{R(a_{oi1}),R(a_{oi2})\}=$ {minimum, maximum}; the naught in the third position may generally be replaced with a 1 or a 2 indicating the limits of the Range; most G-node Ranges consist of two values. Example (for integer type elements): G-node $T_{o5o}$, has G-node Range $R(T_{o5o})=\{R(t_{o51}), R(t_{o52})\}=\{75, 116\}$; this means that G-node $T_{o5o}$ a may contain elements between 75 and 116 in value and still be consistent with the data-structure rules. On diagrams, G-node Ranges are depicted parenthesized under the P-nodes (G-nodes) to which they apply.

G-nodes—a set of P-nodes related by their G-node Ranges. If the processor number subscript and the element number subscript of a set member are naught, then the representation is of a G-node. The set T as a set of G-nodes: $T=\{t_{o1o}, t_{o2o}, t_{o3o}, t_{o4o}\}$The graphical representation of a G-node is rather unique, being distributed amongst processors around the page. FIGS. **1** and **2** contain the same set of data values on a serial b-tree and a parallel b-tree respectively. FIG. **2** contains five (5) G-nodes: $A_{o1o}$, $A_{o2o}$, $B_{o1o}$, $C_{o1o}$, $D_{o1o}$. Assuming the parallel B-tree in FIG. **2** contains the set S of integer data elements, we depict the set S as data-structure nodes, G-nodes, and P-nodes: $S=\{A, B, C, D\}=\{\{A_{o1o}, A_{o2o}\}, \{B_{o1o}\}, \{C_{o1o}\}, \{D_{o1o}\}\}=\{\{\{A_{11o}, A_{21o}\}, \{A_{12o}, A_{220}\}\}, \{B_{11o}, B_{21o}\}, \{C_{11o}, C_{21o}\}, \{D_{11o}, D_{21o}\}\}$. G-node $A_{o2o}$ comprising P-nodes $A_{12o}$ and $A_{22o}$ is identified in FIG. **2**.

P-nodes—assume a number (P>1) of processors: P-nodes assume multiple elements; P-nodes have three(3) subscripts (processor-number, node-number, element or cell number). A P-node has multiple cells for elements; when the element-number subscript is specified, the reference is to a specific cell within the P-node; when the element-number subscript is naught (o), the reference is to the entire P-node. Reference to P-node cells: (on processor**1**) $T=\{t_{111}, t_{112}, t_{121}, t_{122}, t_{131}, t_{132}\}$; Reference to P-nodes: $T=\{t_{110}t_{120}, t_{130}\}$. For greater convenience, P-nodes may be identified by a node letter and non-subscripted processor number (e.g. A1, A2, B1, B2, etc.) See G-nodes.

Sets—S, P, G-nodes and sets of elements. Sets are designated by upper case letters; members of sets are generally designated by lower case, subscripted letters;

S-nodes—nodes within a sequential data-structure. S-nodes in a set generally have only one subscript $S=\{S_1, S_2, S_3, \ldots\}$;

Subscripts—integers or variables between 1 and 9 inclusive, or a naught (o). Subscript numbers will not exceed 9 unless otherwise stated. Naught (o)—represents an absence of specification of individual members of a set with regard to the subscript position, and, thereby, will define a sub-set. Variables with three(3) subscripts have the following subscript order: $S_{ijk}$: $S_{processor\ number,\ node\ number,\ cell\ number}$

Preferred Embodiment

Preferences for Adaptable Data-Structures

1). Ordinal data is preferred for the elements contained and ordered within the sequential data-structure by the sequential method and/or rules of ordering.

2). The sequential data-structure is preferred to be capable of representation by nodes containing data elements and links that relate the nodes according to the relationships of the elements contained in the nodes. The relationships represented by the links may relate the node to adjacent nodes, non-adjacent nodes and/or to the rest of the data-structure as a whole. 3). The adapted nodes are preferred to have the capability of the calculation of G-node Ranges which may be related to each other in an ordinal fashion. 4). Contiguity: the structure is preferred to have the quality that the placement of nodes makes the data ranges contiguous with respect to the structure and rules of the graph and the data set contained and organized according to its ranges.

Preferred Embodiment

General Description

The purpose of this section is to describe the data-structures and functions in a less technical manner than that of the pseudo-code contained in other sections.

This section contains only a generalized description of the present method and does not contain all the details of the invention. For ease of understanding, the description in this section is presented using graphical depictions of the data-structures in their sequential (single-processor) forms along with accompanying descriptions of the graphical figures; the parallelized forms of the data-structures are then depicted in the same fashion. It may serve as an introduction to the basic concepts of the present method so that the reader will find the other descriptions easier to follow.

The problem that the invention solves is presented in the section Discussion of Problem. The example given in that section functions in the same manner as the examples given here, but the description in this section explains the underlying functionality that produces the results shown in the previous section. In addition, two Examples are shown here to ensure that the general concept is understood to apply to various types of data-structures. A complete description of the present method, the basic parallel method, its functionality and the data-structures that it produces are described in other sections.

Preferred Embodiment

Description of the Shapes of Single-processor Data-Structures and their Multi-processor Counter-parts

This section describes the configuration of adapted parallel data-structures, how they are stored on multiple processors or memory storage devices, and how they are similar to their single-processor counter-parts. The values of the data elements in a single-processor data-structure determine its shape. [see single B-tree FIG. **1**] The single-processor B-tree in FIG. **1** contains 12 distinct values and has 12 distinct positions for those values. The numerical relationships (greater-than/less-than) between the 12 elements in the B-tree in FIG. **1** determine the shape of the tree and the positions of the elements.

The values of the data elements in a multi-processor data-structure also determine its shape; however, they determine its shape according to the ranges of values into which they fall [see parallel B-tree FIG. **2**]. Although the same 12 elements populate the parallel B-tree in FIG. **2**, the shape of the parallel B-tree is not determined by the positions of 12 distinct elements, but by the positions of 5 distinct ranges that the 12 elements fall into.

Comparing the trees in FIG. **1** and FIG. **2**, we see that the elements 20 and 26 occupy node C in FIG. **1**. The contents of node C are determined by the fact that the parent node A of the tree contains the two values 15 and 30: therefore all elements greater than 15 and less than 30 are placed in node C. The Adapted parallel version of the tree in FIG. **2** also has a node C; however, the parallel node C has two parts **C1** (on processor **1**) and **C2** (on processor **2**) . The contents of the parallel node C are determined by the fact that the parallel parent node A contains two ranges of values (15 to 20) and (40 to 45): therefore all elements greater than (15 to 20) and less than (40 to 45) are placed in parallel node C. Therefore the elements in parallel node C fall into the range (21 to 39); these elements are 26,30 and 33.

The parallel B-tree in FIG. **2** is composed of two identically shaped trees (one on each processor). The elements in these identical trees are also positioned identically within the tree according to the ranges that they fall into. This grouping of elements according to identical ranges on each processor creates a Global-node or G-node: the G-node is a collection of data elements in identical positions within identical data-structures contained on multiple processors or processing components. Each G-node has its range (G-node Range) recorded on each processor. The G-node with G-node range (40 to 45) is positioned as the second entry in node A in FIG. **2**. If the value 43 were Inserted by either processor into the parallel B-tree, then it would take position in this G-node because it falls into the G-node range (40–45). This G-node would then contain the values 40,43, and 45. The concept of the G-node is central to the functioning of the parallelized method/data-structure: once the concepts of the G-node, the G-node Range and the G-node Split are firmly grasped, the present method should be fairly easy to comprehend. The G-node Split is explained in the following section.

## Preferred Embodiment

### Verbal Description (Insert and Remove)

This section gives a verbal description of how the preferred embodiment functions. Adapted parallel data-structures created by the preferred embodiment are always composed of P identical data-structures, each contained on one of P processors or system components. The adapted parallel data-structures take form and are organized according to the same principles (with respect to G-node Ranges) that form and organize the single-processor data-structures from which the parallel versions are derived.

As mentioned previously, the single-processor data-structures to be adapted are created and maintained through the use of Insert and Remove functions for the respective data-structures. The ability to Insert and Remove from ordered lists of data implies the ability to search. Search (Find) functions are preferred aspects of the single-processor Insert and Remove functions in general.

The multi-processor Insert, Remove and Find functions may be originated at any time, on any processor (1 to P). The processor originating the Insert, Remove or Find function may or may not need to involve the other processors in the effort. In some cases these functions can be executed by a single processor within the parallel or distributed system. Whether or not other processors need to be involved, depends on how much room there is in a G-node for Insert and whether or not a specific value is present on a given processor for Find or Remove.

For this general description, the parallel versions of the Insert and Remove functions may be said to have three phases: (1) Location of the proper G-node on the originating processor (2) Location of the proper processor (1 through P) with insertion or removal of the element on that processor (3) Performance of G-node Split or G-node deletion if necessary. Step 1 can be performed by any single-processor, at any time, independent of the other processors. Step 2 involves more than one processor in a cooperative effort unless the "proper processor" is the processor that originated the Insert or Remove. Step 3 usually requires all processors to communicate for a G-node Split because the elements in the G-node must be sorted across processors for a Split; Step 3 usually does not require all processors to communicate for a G-node deletion.

The following steps 1 through 3 are also identified in the pseudo-code for the parallel Insert and Remove functions given in the Program Adaptation section.

Step 1

(Location of the proper G-node on the originating processor (Find G-node))

The functioning of the parallelized method depends on the functioning of the single-processor method. The single-processor method functions according to the relationships between the values of the elements: the multi-processor method functions according to the relationships of the ranges of values of the elements.

The search of an ordered list is performed by comparing the values found at positions within the data-structure. For Example: Searching the single-processor B-tree for 33 in FIG. **1**, we start at the top node and compare the values. 33 falls between 30 and 45, so we travel down the link under 45 and find node D. Searching node D from left to right we immediately locate 33. Searching the multi- processor B-tree for 33 in FIG. **2**, we start at the top node and compare the ranges. 33 falls between the ranges (15 to 20) and (40 to 45), so we travel down the link under (40 to 45) and find node C. Parallel node C may be located by either processor **1** or processor 2. Searching parallel node C for the value 33 is described in Step 2.

Step 2

(Location of the proper processor (1 through P)) Once a given processor p has successfully located the proper G-node within its data-structure, it may then send the location of this G-node to the other processors in the system. Each of these processors may then attempt to locate the search value within its own portion of the G-node or attempt to place a value in the proper G-node.

In Step 1 above, we located G-node C (FIG. **2**) as the proper node for **33**. If the originating processor is processor **1**, it sends a request to processor **2** to search G-node C; processor **2** then searches its portion of G-node and finds the value; it may then Insert or Remove the value from the data-structure. If the originating processor is processor **2**, it immediately locates the value 33 and need not make any request of processor **1**.

Whether or not the originating processor needs to send requests to other processors for location of values is dependent on the ordering of values within the G-node. The data-structures in FIG. 2 have G-nodes with unordered internal values. For a discussion on ordering values within G-nodes, see other sections.

Step 3

(Performance of G-node Split or G-node deletion if necessary) A G-node Split is the creation of a new G-node;

(83)

a G-node deletion is the destruction of an existing G-node. When a G-node is considered full, it is Split; when it is considered empty (or sufficiently empty), it is destroyed and deleted from the data-structure. The fullness or emptiness of a G-node is similar in conception to the fullness or emptiness of a node in a single-processor B-tree or m-way search tree. The Rule for Fullness in this section is set forth in detail below.

The G-node Split is described in the definition section above; this definition is sufficient for the General Description. For a more detailed description, see the Function Explanation sections. The G-node deletion is simply the removal of the G-node from the data-structure.

Once a G-node is created or destroyed, it must be added or deleted from the given data-structure according to the Rules for the data-structure with respect to the G-node Ranges. Examining the data-structure on processor 1 in FIG. 2, we see that it is a valid B-tree, in its own right, regardless of the existence of other processors: if we remove the value 5 from this serial B-tree, we produce the B-tree on processor 1 in FIG. 3. When both processor 1 and 2 perform this removal simultaneously, each processor redistributes the B-tree according to the rules of B-tree configuration: the result is a G-node deletion. Note that this would require the absence of all three values in the G-node: 4, 5, and 12. The point being made here is that G-node additions and deletions function according to the same rules as the single-processor data-structures. This process is clarified further in the two examples in the following section.

## Preferred Embodiment

### Descriptions by Example

A single-processor method creates its data-structure (such as a B-tree) by Inserting and Removing the values contained in the list to be maintained according to the Rules of the Insert and Remove functions for the data-structure. The Method of Adapting single-processor methods and their associated data-structures into multi-processor methods and data-structures makes use of the single-processor method. Each of the following two Examples will create precisely the same configurations for their data-structures in both the single and multi-processor versions described. A reader understanding the functioning of AVL trees and B-trees should be able to see the functioning of the multi-processor method as a transformation of the single-processor method in each case. The present method transforms the single-processor method into the multi-processor method.

Any implementation of a parallelized data-structure may utilize variations on the Rules for Fullness of G-nodes and the Ordering Scheme of elements within G-nodes. The following rules and ordering scheme will be used for these two Examples:

1. Rules for fullness/emptiness: each G-node in these Examples will be composed of 3 sets of elements; each set will contain zero through two elements. A G-node i is full when all three sets contain two elements (the G-node therefore containing six elements). The G-node is empty when all three sets contain zero elements.
2. Ordering Scheme: each P-node set within a G-node is contained on a single-processor (P1, P2 or P3 ). The elements in the G-node will be kept in ascending order across the processors and evenly distributed (P1 containing the smallest values, P2 the mid-most, P3 the largest).

The following two Examples explain the underlying functioning of the preferred embodiment by maintaining the same list of values on two parallelized data-structures. Each Example first describes the functioning of the single-processor version of the data-structure on a similar list. Each Example then describes the parallel version of the data-structure. These Examples both use three (P=3) processors for the parallel data-structures.

The single-processor versions of the lists are roughly one-third the size of the parallel versions. Each single value inserted or deleted in the single-processor data-structure is matched by several values for the parallel version. The multiple values inserted and deleted in the parallel version are specifically chosen to fall into the proper G-node Ranges so that the single and multi-processor data-structures take on the same configurations: this is done so that the identical functioning, form and structure of the single and multi-processor versions can be easily seen. The functioning of the parallel versions is in no way dependent on any choice of element values (any list of ordinable data elements may be Inserted or Removed in any order).

### EXAMPLE 1

Single-processor Method

The single-processor AVL tree method is composed of finding the proper location for a new node, adding that node, and performing rotation.

Example 1 begins with FIG. 4, showing a properly ordered single-processor AVL tree containing the elements from the single-processor initial list.

1.) Insert(60)

Comparing values at each node: Root-node A: [60>40], travel down the right-most link to node C; node C: [60>50], travel right to node F; node F: [60<70]—F has no left link so we create a new node G and place it to the left of node F. (FIG. 5)

Node G has been added in its proper place; the AVL tree is left unbalanced, therefore we perform RL rotation (FIG. 6).

2.) Insert(80)

Root-node A: [80>40] travel right to node G; node G: [80>60] travel right to node F; node F: [80>70]—F has no right link so we create a new node H and place it to the right of node F. The tree is still balanced (no rotation) (FIG. 7).

3.) Insert(90)

Root-node A: [90>40], travel right to node G; node G: [90>60], travel right to node F; node F:[90>70], travel right to node H; node H: [90>80]—H has no right link, so we create a new node I and place it to the right of H (FIG. 8).

Node I has been added in its proper place; the AVL tree is left unbalanced, therefore we perform RR rotation (FIG. 9).

4.) Remove(60)

Root-node A: [60>40], travel right to node G; node G: [60=60], therefore delete node G; replace node G with the left-most child of the right sub-tree (node F). The tree is still balanced (no rotation) (FIG. 10).

Multi-processor Method

The multi-processor AVL tree method is composed of finding the proper location for a new value, inserting the values until a G-node Split thereby creating a new G-node, adding that G-node, and then performing rotation in parallel. Refer to Steps 1 through 3 in the Verbal Description section.

Example 1 (multi-processor) begins with FIG. 11, showing an Adapted AVL tree composed of 3 properly ordered AVL trees on 3 processors containing the elements from the

(121)

multi-processor initial list. The G-node Ranges are shown beneath the parallel nodes.

1.) Insert(60) [after which Insert(65)]

Insert(60) at processor P1

Step 1

Comparing values at each G-node:

Root-node A1: [(60)>(40–49)], travel down the right-most link to node C1; node C1: [(60)>(50–59)], travel right to node F1; node F1: [(60)=(60-max)]—60 falls within G-node Range (60-max), so we add 60 to this G-node at processor P1.

Step 2

The values are properly ordered within the G-node, so step 2 is not necessary.

Step 3

The G-node is NOT full, so Step 3 is not necessary (no G-node Split). (FIG. 12)

Insert (65) at Processor P1

Step 1

Root-node A1: [(65)>(40–49)] travel right to node C1; node C1: [(65)>(50–59)] travel right to node F1; node F1: [(65)=(60-max)]—65 falls within G-node Range (60-max), so we add 65 to this G-node at processor P1. (FIG. 13)

Step 2

As FIG. 13 shows, the G-node F1 at processor P1 has 3 values. This is greater than maximum number of values per processor per G-node, so we perform Step 2 to properly order the values within the G-node in F. Processor P1 sends the value 70 to P2, and P2 sends 75 to P3. This exchange of elements maintains the Ordering Scheme within G-nodes, locating the proper processors for each value (rule 2 listed above) (FIG. 14).

Step 3:

After addition of 65 to this G-node, the G-node is full, therefore we perform a G-node Split. The G-node Split divides the values 60,65,70,74,75,78 into two ranges. The lower range is placed into a newly created G-node (in G), the upper range is kept in the G-node in F (FIG. 15).

The new G-node (in G) is placed to the left of the G-node in F because its G-node Range (60–70) is less than the G-node Range of F (71-max) (FIG. 16).

The addition of the new node F leaves the AVL tree unbalanced as it did in the single-processor example, therefore we perform RL rotation in parallel (FIG. 17). 2.) Insert(80),Insert(89),Insert(85)

These three Insertions are performed simultaneously. Because the Insertions at processors P1, P2, and P3 all follow the same procedure, we will follow Step 1 only at processor P2.

Insert(80) at P1, Insert(89) at P2, Insert(85) at P3

Step 1 at Processor P2

Root-node A2: [(89)>(40–49)], travel right to node G2; node G2: [(89)>(60–70)], travel right to node F2; node F2: [(89)=(71-max)]—89 falls within G-node Range (71-max), so we add 89 to this G-node at processor P2. Following identical comparisons at processors P1 and P3, the values 80 and 85 have been added to the same G-node (FIG. 18).

Step 2:

As FIG. 18 shows, the values are not arranged in ascending order within the G-node in F, so we perform Step 2. Processor P1 sends the value 80 to P2, and P2 sends 75 to P1 and also 89 to P3, P3 sends 78 to P2. This exchange of elements maintains the Ordering Scheme within G-nodes, locating the proper processors for each value (rule 2 listed above) (FIG. 19).

Step 3:

After addition of 80, 89 and 85 to this G-node, the G-node is full, therefore we perform a G-node Split. The G-node Split divides the values 74,75,78,80,85,89 into two ranges. The lower range is placed into a newly created G-node which is in the AVL tree node (F), the upper range is kept in the G-node in the newly formed AVL tree node (H) (FIG. 20).

The AVL tree node (H) containing the old G-node is placed to right of node F because its G-node Range (79-max) is greater than the G-node Range of F (71–78) (FIG. 21). The addition of the new node H leaves the AVL tree balanced (no rotation).

3.) Insert(98),Insert(95),Insert(90)

These three Insertions are performed simultaneously. Because the Insertions at processors P1, P2, and P3 all follow the same procedure, we will follow Step 1 only at processor P3.

Insert(98) at P1, Insert(95) at P2, Insert(90) at P3

Step 1 at Processor P3

Root-node A3: [(90)>(40–49)], travel right to node G3; node G3: [(90)>(60–70)], travel right to node F3; node F3: [(90)>(71–78)], travel right to node H3; node H3: [(90)=(79-max)]—90 falls within G-node Range (79-max), so we add 90 to this G-node at processor P3. Following identical comparisons at processors P1 and P2, the values 98 and 95 have been added to the same G-node (FIG. 22).

Step 2

As FIG. 22 shows, the values are not arranged in ascending order within the G-node in H, so we perform Step 2. Processor P1 sends the value 98 to P3, and P2 sends 85 to P1 and also 95 to P3, P3 sends 89 and 90 to P2. This exchange of elements maintains the Ordering Scheme within G-nodes, locating the proper processors for each value (rule 2 listed above) (FIG. 23).

Step 3

After addition of 98, 95 and 90 to this G-node, the G-node is full, therefore we perform a G-node Split. The G-node Split divides the values 80,85,89,90,95,98 into two ranges. The upper range is placed into a newly created G-node (I), the lower range is kept in the G-node in H (FIG. 24).

The new G-node (I) is placed to right of G-node H because its G-node Range (90-max) is greater than the G-node Range of H (79–89) (FIG. 25). The addition of the new G-node I leaves the AVL tree unbalanced as it did in the single-processor example, therefore we perform RR rotation in parallel (FIG. 26).

4.) Remove(55), Remove(65), Remove(70), [after which Remove(60)]

Remove(55) at P1, Remove(65) at P2, Remove(70) at P3 (performed simultaneously)

Remove(55) at P1

Step 1

Root-node A1: [(55)>(40–49)], travel right to node G1; node G1: [(55)<(60–70], travel left to node C1: node C1: [(55)=(50–59)]—55 falls within the G-node Range (50–59), so processor P1 looks in node C1 for the value 55. 55 is not in node C1 at processor P1, so we must perform Step 2.

Step 2

P1 send a request to the other processors to look for 55 in their respective nodes C2 and C3. Processor P2 finds 55 in its node. P2 removes the value 55 from node C2 and sends it to P1 (FIG. 27).

Step 3

G-node C is not empty and so Step 3 is not necessary.

Remove(65) at P2

Root-node A2: [(65)>(40–49)], travel right to node G2; node G2: [(65)=(60–70)]—65 falls within the G-node

Range (60–70), so processor P2 looks in node G2 for the value 65 and finds 65 in G2.

P2 then removes 65 from node G2.

Step 2

The value that P2 searched for (65) was found at P2, therefore Step 2 is not necessary (FIG. 27).

Step 3

G-node G is not empty and so Step 3 is not necessary.

Remove(70) at P3

Root-node A3: [(70)>(40–49)], travel right to node G3; node G3: [(70)=(60–70)]—70 falls within the G-node Range (60–70), so processor P3 looks in node G3 for the value 70 and finds 70 in G3.

P3 then removes 70 from node G3.

Step 2

The value that P3 searched for (70) was found at P3, therefore Step 2 is not necessary (FIG. 27).

Step 3

The G-node in G is not empty and so Step 3 is not necessary.

Remove(60) at P2

Root-node A2: [(60)>(40–49)], travel right to node G2; node G2: [(60=(60–70)]—60 falls within the G-node Range (60–70), so processor P2 looks in node G2 for the value 60. 60 is not in node G2 at processor P2, so we must perform Step 2.

Step 2

P2 sends a request to the other processors to look for 60 in their respective nodes G1 and G3. Processor P1 finds 60 in its node G1. P1 removes the value 60 from node G1 and sends it to P2 (FIG. 28).

Step 3

The G-node in G is empty and so we perform Step 3. The removal of the G-node G is simply a matter of each of the processors P1, P2 and P3 performing a normal AVL node removal. P1 removes G1 from its tree; P2 removes G2 from its tree; P3 removes G3 from its tree. Each of the processors re-orders the tree according to the single-processor AVL tree method and replaces node G with the left-most child of the right sub-tree and performs range adjustment (node F) (FIG. 29).

EXAMPLE 2

Single-processor Method

The single-processor B-tree method is composed of finding the proper location for a new value, adding that value, and performing B-tree splits when the B-tree nodes are full (contain 3 values).

Example 2 begins with FIG. 30, showing a properly ordered single-processor B-tree (degree 3) containing the elements from the single-processor initial list.

1.) Insert(60)

Comparing values at each node, moving through the tuples from left to right:

Root-node A: [60>20], move right; [60>40], travel down the right-most link to node D; node D: insert 60 between 50 and 70 at node D (FIG. 31).

D now has 3 values and must be split. The right-most value goes in the new node (E). The left most value is kept in node D; the middle value (60) becomes the parent value of D and is re-inserted at the parent node A (FIG. 32).

The parent node A (root-node) now has 3 values and must be split. The right-most value goes in the new node (F). The left most value is kept in node A; the middle value (40) becomes the parent value of A and is re-inserted at the parent (no parent exists for the root, so a new root is created—node G) (FIG. 33).

2.) Insert(80)

Root-node G: [80>40] travel right to node F; node F: [80>60] travel right to node E; node E: insert 80 after 70 at node E (FIG. 34).

3.) Insert(90)

Root-node G: [90>40], travel right to node F; node F: [90>60], travel right to node E; node E: insert 90 after 80 at node E (FIG. 35). Node E now has 3 values and must be split. The right most value goes in the new node (H). The left most value is kept in node E; the middle value (80) becomes the parent value of E and is re-inserted at the parent node F (FIG. 36).

4.) Remove(60)

Root-node G: [60>40], travel right to node F; node F: 60 is found at node F and removed. This leaves F with too few values, so it removes node E, places its value (70) in node D, and makes 80 the parent value of node D (FIG. 37).

Multi-processor Method

The multi-processor B-tree method is composed of finding the proper location for a new value, inserting the values until a G-node Split thereby creating a new G-node, adding that G-node, and performing B-tree splits when the B-tree nodes are full (contain 3 G-nodes). (The G-nodes constitute the elements of the B-tree.)

Refer to Steps 1 through 3 in the Verbal Description Section

Example 2 (multi-processor) begins with FIG. 38, showing an Adapted B-tree composed of 3 properly ordered B-trees on 3 processors containing the elements from the multi-processor initial list. The G-node Ranges are shown beneath the parallel G-nodes.

1.) Insert(60) [after which Insert(65)]

Insert(60) at processor P1

Step 1

Comparing values at each node, moving through the tuples from left to right:

Root-node A1:[(60)>(20–29)], move right; [(60)>(40–49)], travel down the right-most link to node D1; node D1: insert 60 into right-most G-node in node D.

Step 2

The values are properly ordered within the G-node, so step 2 is not necessary.

Step 3

The G-node is NOT full, so Step 3 is not necessary (no G-node Split). (Note that although node D1 has three values, it contains only 2 G-nodes and therefore does not need a B-tree split.) (FIG. 39)

Insert (65) at processor P1

Step 1

Root-node A1: [(65)>(20–29)] move right; [(65)>(40–49)], travel down right-most link to node D1; node D1: insert 65 into second G-node in node D (FIG. 40).

Step 2

As FIG. 40 shows, the second G-node in D1 at processor P1 has 3 values. This is greater than maximum number of values per processor per G-node, so we perform Step 2 to properly order the values within the G-node, Processor P1 sends the value 70 to P2, and P2 sends 75 to P3. This exchange of elements maintains the Ordering Scheme within G-nodes, locating the proper processors for each value (rule 2 listed above) (FIG. 41).

Step 3

After addition of 65 to this G-node, the G-node is full, therefore we perform a G-node Split. The G-node Split divides the values 60,65,70,74,75,78 into two ranges. The lower range is placed into a newly created G-node, the upper range is kept in the existing G-node (FIG. 15).

The new G-node is placed to left of the existing G-node because its G-node Range (60–70) is less than the G-node

Range (71-max) (FIG. 42). D now contains 3 G-nodes and must be split (B-tree split). The right most G-node goes in the new B-tree node (E). The left most G-node is kept in B-tree node D; the middle G-node with G-node Range (60–70) becomes the parent value of D and is re-inserted at the parent node A (FIG. 43). The parent node A (B-tree-root-node) now has 3 G-nodes and must be split. The right most G-node goes in the new B-tree node (F). The left most G-node is kept in node A; the middle G-node (40–49) becomes the parent value of A and is re-inserted at the parent (no parent exists for the root, so a new root is created—B-tree node G) (FIG. 44).

2.) Insert(80),Insert(89),Insert(85)

These three Insertions are performed simultaneously. Because the Insertions at processors P1, P2, and P3 all follow the same procedure, we will follow Step 1 only at processor P2.

Insert(80) at P1, Insert(89) at P2, Insert(85) at P3 Step 1 at processor P2:

Root-node G2: [(89)>(40–49)], travel right to node F2; node F2: [(89)>(60–70)], travel right to node E2; node E2: [(89)=(71-max)]—89 falls within G-node Range (71-max), so we add 89 to this G-node at processor P2. Following identical comparisons at processors P1 and P3, the values 80 and 85 have been added to the same G-node (FIG. 45).

Step 2

As FIG. 45 shows, the values are not arranged in ascending order within the G-node at E, so we perform Step 2. Processor P1 sends the value 80 to P2, and P2 sends 75 to P1 and also 89 to P3, P3 sends 78 to P2. This exchange of elements maintains the Ordering Scheme within G-nodes, locating the proper processors for each value (rule 2 listed above) (FIG. 46).

Step 3

After addition of 80, 89 and 85 to this G-node, the G-node is full, therefore we perform a G-node Split. The G-node Split divides the values 74,75,78,80,85,89 into two ranges. The lower range is placed into a newly created G-node, the upper range is kept in the existing G-node (FIG. 20).

The new G-node is placed to left of the existing G-node because its G-node Range (71–78) is less than the G-node Range (79-max) (FIG. 47).

3.) Insert(98),Insert(95),Insert(90)

These three Insertions are performed simultaneously. Because the Insertions at processors P1, P2, and P3 all follow the same procedure, we will follow Step 1 only at processor P3.

Insert(98) at P1, Insert(95) at P2, Insert(90) at P3

Step 1 at processor P3:

Root-node G3: [(90)>(40–49)], travel right to node F3; node F3: [(90)>(60–70)], travel right to node E3; node E3: [(90)>(71–78)], move right; [(90)=(79-max)]—90 falls within G-node Range (79-max), so we add 90 to this G-node at processor P3. Following identical comparisons at processors P1 and P2, the values 98 and 95 have been added to the same G-node (FIG. 48).

Step 2

As FIG. 48 shows, the values are not arranged in ascending order within the G-node at E, so we perform Step 2. Processor P1 sends the value 98 to P3, and P2 sends 85 to P1 and also 95 to P3, P3 sends 89 and 90 to P2. This exchange of elements maintains the Ordering Scheme within G-nodes, locating the proper processors for each value (rule 2 listed above) (FIG. 49).

Step 3

After addition of 98, 95 and 90 to this G-node, the G-node is full, therefore we perform a G-node Split. The G-node Split divides the values 80,85,89,90,95,98 into two ranges. The lower range is placed into a newly created G-node, the upper range is kept in the existing G-node (FIG. 24).

The new G-node is placed to left of the existing G-node because its G-node Range (79–89) is less than the G-node Range (90-max) (FIG. 50). Node E now has 3 G-nodes and must be split (B-tree split). The right most G-node goes in the new node (H). The left most G-node is kept in node E; the middle G-node (79–89) becomes the parent G-node of E and is re-inserted at the parent node F (FIG. 51).

4.) Remove(55), Remove(65), Remove(70), [after which Remove(60)]

Remove(55) at P1, Remove(65) at P2, Remove(70) at P3 (performed simultaneously)

Remove(55) at P1

Step 1

Root-node G1: [(55)>(40–49)], travel right to node F1; node F1: [(55)<(60–70], travel left to node D1: node D1: [(55)=(50–59)]—55 falls within the G-node Range (50–59), so processor P1 looks in node D1 for the value 55. 55 is not in node D1 at processor P1, so we must perform Step 2.

Step 2

P1 send a request to the other processors to look for 55 in their respective nodes D2 and D3. Processor P2 finds 55 in its node D2. P2 removes the value 55 from node D2 and sends it to P1 (FIG. 52).

Step 3

The G-node in D is not empty and so Step 3 is not necessary.

Remove (65) at P2 Root-node G2: [(65)>(40–49)], travel right to node F2; node F2: [(65)=(60–70)]—65 falls within the G-node Range (60–70), so processor P2 looks in node F2 for the value 65 and finds 65 in F2. P2 then removes 65 from node F2.

Step 2

The value that P2 searched for (65) was found at P2, therefore Step 2 is not necessary (FIG. 52).

Step 3

The G-node in F is not empty and so Step 3 is not necessary.

Remove(70) at P3

Root-node G3: [(70)>(40–49)], travel right to node F3; node F3: [(70)=(60–70)]—70 falls within the G-node Range (60–70), so processor P3 looks in node F3 for the value 70 and finds 70 in F3. P3 then removes 70 from node F3.

Step 2

The value that P3 searched for (70) was found at P3, therefore Step 2 is not necessary (FIG. 52).

Step 3

The G-node in F is not empty and so Step 3 is not necessary.

Remove(60) at P2

Root-node G2:[(60)>(40–49)], travel right to node F2; node F2: [(60=(60–70)]—60 falls within the G-node Range (60–70), so processor P2 looks in node F2 for the value 60. 60 is not in node F2 at processor P2, so we must perform Step 2.

Step 2

P2 sends a request to the other processors to look for 60 in their respective nodes F1 and F3. Processor P1 finds 60 in its node F1. P1 removes the value 60 from node F1 and sends it to P2 (FIG. 53).

Step 3

The G-node in F is empty and so we perform Step 3. The removal of the G-node is simply a matter of each of the

processors P1, P2 and P3 performing a normal B-tree-node removal. P1 removes the G-node from F1 in its tree; P2 removes from F2; P3 removes from F3. Each of the processors re-orders the tree according to the single-processor B-tree method and removes node E, places its G-node (71–78) in node D, makes (79–89) the parent value of node D and performs range adjustment (FIG. 54).

Preferred Embodiment

Rules for Fullness and Ordering Scheme for B-trees Stored on Disk

The usage of Rules for Fullness and Ordering Schemes is described in the previous sections. The Rule for Fullness and Ordering Scheme chosen for those examples assume that the parallel data-structure is not stored on disk. A different rule and scheme should be chosen if the processing-elements of the parallel data-structure are disk-packs rather than actual CPU's. It should also be noted here that the terms "processor" and "processing-element" are used to refer to system components to which work may distributed in the maintenance of the parallel data-structure: in this section the processing-elements are assumed to be disk-packs on a system with multiple disk drives; the work distributed amongst the disk-packs is the actual reading and writing of the blocks that contain the parallel B-tree-nodes.

In this section, another Example of a parallel data-structure is given. The purpose of the example is to illustrate the functionality of the Rules for Fullness and Ordering Scheme chosen for the B-tree stored on disk. The example describes one possible embodiment of an adapted B-tree. The manner of describing this example is the same as the manner used in the previous sections.

The main difference between storing data in memory and on disk is that disk access is slower. Assuming that the location of the memory block or disk block is known, accessing data on disk might take milli-seconds whereas accessing data in memory would take only micro-seconds. Therefore, the goal in designing data-structures to be stored on disk is to minimize the number of disk accesses necessary to locate the desired data-block. The goal in the design of the parallel data-structures described in this invention is to allow the same data-structure to be accessed simultaneously by multiple processing-elements (or disk-packs in this section) and thus distribute the work amongst the processing-elements. Because the goal in designing data-structures for disk is to minimize accesses, the Rule for Fullness and the Ordering Scheme of a disk-stored parallel B-tree must be defined to minimize parallel communication between processing-elements (disk-packs) and provide the most efficient access paths possible to desired P-nodes. Steps 2 and 3 described in the Verbal Description require parallel communication: the parallel communication in Step 2 can be minimized by choosing an Ordering Scheme that does not involve all of the disk-packs in locating the proper disk-pack for placement of a value. The Rule for Fullness can also be altered so that determining the fullness or emptiness of a G-node does not involve all of the disks.

The following Rule for Fullness and Ordering Scheme will be used in the example for this section:

1. Rule for Fullness/Emptiness: The fullness of a G-node in this Example is dependent on the fullness of the B-tree node that contains the G-node. A B-tree node is considered full when it contains five values (integers) and is thereby ready to undergo a B-tree split. A G-node may be considered full when one-half of the B-tree-nodes that contain the G-node are ready for a B-tree split; a G-node may be considered empty when one-half of the B-tree-nodes that contain it are ready for a merge or deletion. This information can be stored for each parallel B-tree-node outside of the parallel data-structure (possibly in memory). Once one-half of the parallel B-tree-nodes are ready to split, one of the G-nodes within the B-tree-nodes is split.
2. Ordering Scheme: This example uses three disk-packs. Disk 1 will contain the bottom (smallest) one-third of the range of values in a given G-node; Disk 2 will contain the middle one-third; Disk 3 will contain the top (largest) one-third of the Range. (If the G-node Range were (1–100), then Disk 1 would contain any value between 1 and 34; Disk 2 would contain any values between 35 and 67; Disk 3 would contain values between 68 and 100).

The Rule for Fullness/Emptiness above minimizes the need to access all portions of the B-tree-node in question because the information for determining the fullness of the parallel B-tree-node is stored external to the tree. The Ordering Scheme above minimizes the need to access all portions of the B-tree-node in question because the location of the proper Disk for a given value in a given Range can be calculated mathematically: this allows the direct location within memory storage of the exact individual node (P-node) contained in a given G-node that could contain a given data value within the G-node's G-node Range. This Example begins with FIG. 55 showing a parallel B-tree ordered according to Rules 1 and 2 above. Note that although the same values are stored in the tree in FIG. 55 as those stored in FIG. 38, the right-most G-node in the tree is ordered differently: this is because of the Ordering Scheme rule above. At the beginning of this Example none of the B-tree-nodes located in the data-structure are ready to be split.

We now proceed to Insert a number of values into the disk-stored B-tree in FIG. 55.

1.) Insert(60) on Disk 1 and Insert(71) on Disk 2 Simultaneously

Step 1

Root-node A1: [(60)>(20–29)], move right; [(60)>(40–49)], travel down the right most link to D1; node D1: insert 60 into right most G-node in D1. (Disk 2 follows the same pattern)

Step 2

The values are properly ordered within the G-node, so Step 2 is unnecessary.

Step 3

The G-node is not full (no G-node Split)(FIG. 56)

2.) Insert(52) at Disk 1, Insert(51) at Disk 2, Insert(59) at Disk 3

Step 1

(Step 1 is followed at Disk 2 in order to illustrate the functionality of the Ordering Scheme)

Root-node A2: [(51)>(20–29)], move right; [(51)>(40–49)], travel down right-most link to node D2; node D2: the value 51 belongs in the G-node with Range (50–59).

Step 2

Send the value 51 to Disk 1 and place it in the G-node with Range (50–59). According to the Ordering Scheme, 51 must be sent to Disk 1 because it is in the bottom one-third of the Range (50–59). Note that Disk 3 is not involved in Step 2 because the values contained in node D3 play no part in determining the proper Disk for 51: one disk access is saved by the Ordering Scheme.

Step 3

B-tree-node D3 now contains five (5) values because of the insertion of the value 59. The addition of the fifth value

and resulting fullness of the B-tree-node D3 is recorded. Nodes D1 and D2 are still less than full; therefore less than one-half of the parallel nodes are full, and there is no Split—Step 3 is unnecessary at this time. (FIG. **57**)

3.) Insert(53) at Disk **1**

Step 1

The pattern of locating the proper B-tree- node has been well established at this point—see other examples. The correct G-node for insertion of **53** is the G-node in B-tree-node D1 with Range (50–59).

Step 2

53 falls in the bottom one-third of the Range (50–59); therefore Step 2 is unnecessary (FIG. **58**).

Step 3

The Insertion of 53 into B-tree-node D1 causes D1 to be full. Node D3 is already full. Therefore, more than one-half of the parallel nodes are full, and we must perform a B-tree Split and a G-node Split. This requires accessing the data in node D on all three disks.

Parallel node D contains two G-nodes: one with Range (50–59), the other with Range (60–78)[Max]. The G-node with Range (50–59) contains 8 values; the other G-node contains only 6, so the G-node with (50–59) is chosen for the Split: the two resulting G-nodes have Ranges (50–54) and (55–59) (FIG. **59**). The resulting B-tree-node configuration shows that parallel B-tree-node D contains 3 G-nodes and must be split (B-tree Split). The G-node with range (55–59) must be re-inserted at the Root-node A. All three Disks perform this Step in parallel. Re-insertion of the G-node (55–59) causes the Root-node A to Split (FIGS. **61** and **62**).

Preferred Embodiment

Program Adaptation

The sequential maintenance program to be adapted can be made parallel simply by modifying the S-nodes into P-nodes, grouping the P-nodes into G-nodes (the creation of a P-node is done along with the creation of the G-node that contains it), and then adding a few functions in addition to the original sequential functions. Fullness Rules and Ordering Schemes may be chosen or defined for efficiency. The original sequential functions are used to create and maintain the data-structure configuration: these functions are simply modified to sort, search and arrange according to the relationships between G-node Ranges, rather than the relationships between S-node element values. In the preferred instance, G-node Range R(X)<R(Y) if all of the elements $x_i$ in Range R(X) are less than all elements $y_j$ in G-node Range R(Y): this establishes the relationships between G-nodes in the adapted data-structures. The method of altering algorithms is generally to replace comparisons between x and y in the sequential algorithms with comparisons between R(X) and R(Y) for the parallelized functions.

Function List:

1. Create-G-node (element y)
2. Find-G-node (element y)
3. Search-G-node(G-node v, element y)
4. Add-to-G-node(G-node v, element y)
5. Split-G-node(G-node v)
6. Semi-sort-G-node(G-node v)
7. Adjust-G-node-Ranges(G-node v)
8. Insert-G-node(G-node v)
9. Remove-G-node(G-node v)
10. Resolve-Range-conflict(G-node u, G-node v)
11. Remove-from-G-node(G-node v, element y)

Some of the functions listed above (2, 8 and 9) call the slightly modified sequential functions for a given data-structure. "G-node u,v;" and "element y, z, . . . ", etc. are variable declarations or parameters.

Preferred Embodiment Program Adaptation Function Explanations:

1. Create-G-node(element y). This function creates a G-node by creating one P-node per-processor in the same per-processor location in the data-structure. It places the element y in the P-node of the processor chosen to hold y. Any or all processors may place their own elements $y_i$ in their own P-nodes as well. This function defines the G-node Range: because the new G-node will generally be in an undefined state, the G-node Range may be partially or fully undefined; this is represented in most cases, by the use of MAXVAL and/or MINVAL. If the G-node is the first created in the structure, its Range will generally be R(X)= {MINVAL, MAXVAL} for ordinal data. This function works cooperatively with the other processors. Because G-nodes are composed of P-nodes, this function is a parallel node creation function as well as a global node creation function.
2. Find-G-node(element y). The find global node function is a searching function that locates a G-node with a G-node Range into which the element y falls; this function can provide individual access to each separate graph or data-structure, locating a G-node Range without involving the entire global data-structure. Sequential data-structures that already have Search() functions need only modify those functions to work with G-node Ranges as opposed to element values (using the range function R(G-node)). For sequential data-structures that normally have no Search() functions, knowledge of the sequential data-structure must be used to create a proper Find-G-node() function; in such cases, the G-node found may be one of many possible G-nodes if the Ranges overlap. This function returns the G-node location found. After the G-node location is found and returned, this function may be combined with the Search-G-node() function to provide parallel access to the parallel data-structure.
3. Search-G-node(G-node v, element y). This function searches the G-node v cooperatively for the element y as a parallel access function. G-node v obviously must have a Range capable of holding y. This function may be initiated by a given processor i and then have the other processors return the results of their search to the processor i; thus any one processor may search the entire parallel data-structure for an element y by (1) locating the proper G-node at will in its own separate graph and (2) performing a Search-G-node() in cooperation if necessary thus accessing all of the separate graphs together as parallel data-structure.
4. Add-to-G-node(G-node v, element y). The add to global node function is called after the appropriate G-node for element y has been located. This function inserts the element y into G-node v. This function may arrange the G-node elements in any way desirable for a given data-structure according to Rules for Fullness or Ordering Scheme, or this function may simply place element y in an empty cell in the P-node of the requesting processor that is part of G-node v; if this is not possible, then the requesting processor may cooperate with other processors to find an empty cell in which to place y in G-node v.
5. Split-G-node(G-node v). {returns new G-node} This function calls functions 6 and 7. This function is called

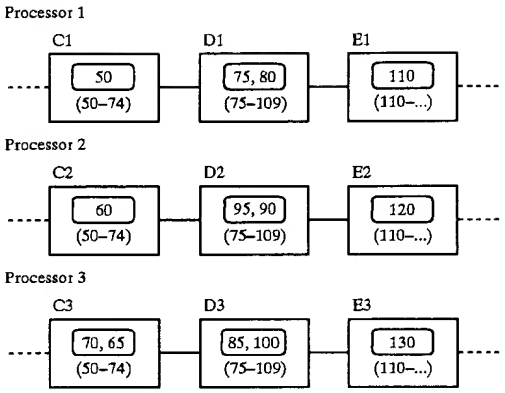
(327)

when G-node v is full. The first step is to call function (6) Semi-sort-G-node() which arranges the elements in G-node v such that they are split into two sets X,Y (X∪Y=W); the resulting sets are partially sorted such that every element $x_i$ falls into a G-node Range distinct from the Range containing all elements $y_i$. Without loss of generality, we assume unique ordinal elements, an ordinal relationship of "less-than," and the preferred method of Range calculation for the data-structure: thus every element $x_i$ is less than every element $y_i$; the set X is contained in cells $v_{iO1}$, the set Y in $v_{iO2}$ (i taking on the values $1 \leqq i \leqq P$). The second step is to create new P-nodes at all processors and move set X or Y into the new P-nodes at each processor i. The third step is to call function (7) Adjust-G-node-Ranges() which resets the G-node Ranges according to the new distribution of elements and creates a new Range for the new G-node. This function (Split-G-node()) may be called on a defined or undefined G-node; after the function ends there will be two G-nodes, one of which will usually remain where it was in the data-structure, the other must be reinserted by function (8) Insert-G-node() or placed appropriately. Generally, if the original G-node v was a defined G-node, then both resulting nodes will be defined; if not, then at least one of the resulting G-nodes will be partially defined. The defined G-node is reinserted (for example see (7) Adjust-G-node Ranges()).

6. Semi-sort-G-node(G-node v). As explained above, this function divides or partially sorts the elements in G-node v and places the resulting distinct sets into the proper processors. This function sub-divides and distributes the portion of data defined by the G-node Range, in essence creating new ranges. The function may also send the minimum and maximum values of the two sets to each processor (or other information for the calculation of Ranges, Fullness, Ordering, etc.).

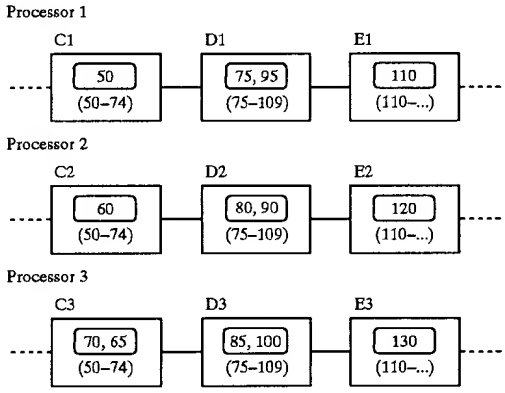
7. Adjust-G-node-Ranges(G-node v). The Adjust-G-node-Ranges() function is key to the adaptation process, performing range determination to group data into value ranges; this function is a range addition and removal function that works in combination with the insert G-node and remove G-node functions. Like the Find-G-node() function it depends on the configuration and rules of the sequential data-structure being adapted. Examples of Split-G-node() and Adjust-G-node-Ranges() are given together because they are so closely related. There are different ways of adjusting Ranges for different data-structures. Also, the G-node v is not the only G-node which will have its Range adjusted; there may be adjustments on any nodes which have their Ranges wholly or partially dependent on G-node v. The goal is as maintain the rule which governs the relationships between nodal values by adjusting the Ranges to fit the new placement of the elements and/or G-node(s). The Adjust-G-node-Ranges() function can operate simultaneously but blindly on all processors. This function may use the minimum and maximum values of the elements of the G-nodes in addition to the values of old Ranges. When adjustments on each processor are made blindly, they are depended upon to be identical over all processors because they use the same values. Example: split and adjustment made in a parallel ordered list with N G-nodes.

Original List

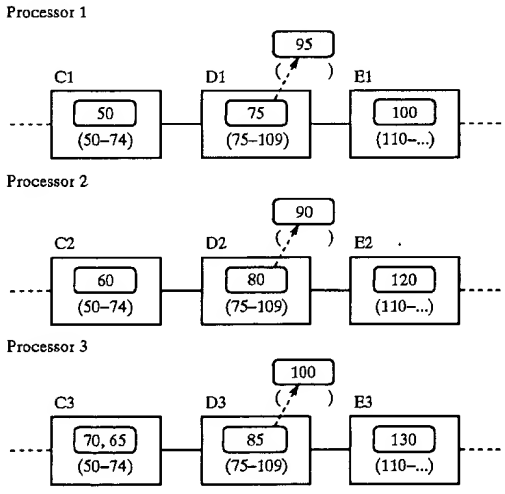



G-node Ranges: $R(C)=R(c_{o1o})=\{50, 74\}R(D)=R(d_{o1o})=\{75, 109\}R(E)=R(e_{o1o})=\{110, \ldots\}$

Step 1: Call Semi-sort-G-node(D)




Step 2: Split G-node D creating G-node V

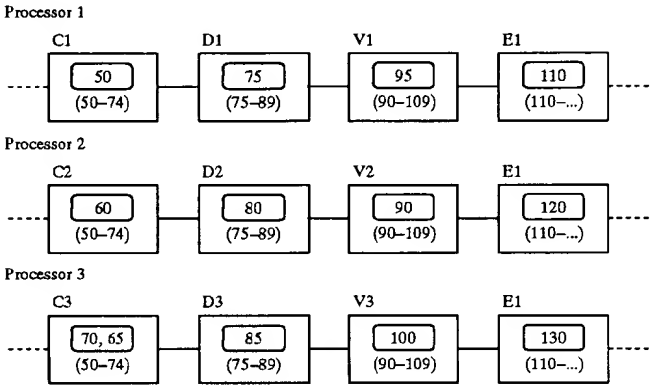


Step 3: Re-insert V (Insert-G-node(V))

In this circumstance (ordered list) the re-insert is predictable and obvious. Step 4 will adjust all G-node Ranges at the same time.

Graph for Step 3

(Note that the Range for G-node V is depicted although it is not calculated until step 4.)




Step 4: Set new G-node Ranges

Under other circumstances, the new Ranges for D would be set as well as for V; after which, insertion of V would take place and a new resetting of Ranges done for the insert; here all Ranges are set at Step 4 because the placement and Range-setting are obvious. However, the pattern should be clear:

Ranges:

R(C): unchanged—{50,74}. R(C)'s second value $R(C_{o12})$ is still based on $R(d_{o11})$which is unchanged.

R(D): $R(d_{o12})$ is changed: {75,89}, $R(d_{o12})=R(v_{o11})-1=90-1=89$.

R(V): {90,109}

$R(v_{o11})$=minimum value of V=90

$R(v_{o12})=R(e_{o11})-1=110-1=109$

R(E) : unchanged.

For this parallel ordered list data-structure, the formula for G-node Ranges R(X) (X taking on values 2<X<(N−1) where N=number of G-nodes) is

$$R(X) = \{ \quad R(x_{o11}) = \text{minimum element of X}, \quad R(x_{o12}) = R((X+1)_{o11} - 1)\}$$

The formulas for other data-structures, though more complex in general, are very similar. At the extreme ends of the spectrum for the parallel ordered list we have special values:

For N G-nodes we have:

$R(a_{o11})$=MINVAL

$R(n_{o12})$=MAXVAL

The G-nodes A and N are partially defined G-nodes.

The above example shows G-node addition, adding one new G-node and one new range to the parallel data-structure. If the new G-node v were then removed, the process would simply be reversed, removing G-node v and removing G-node v's range by merging the range of G-node V into the range of G-node D.

8. Insert-G-node(G-node v). This function works the same as the Insert() function for the sequential data-structure except that it uses the values of G-node Ranges to arrange and relate the G-nodes rather than element values to arrange S-nodes (using the range function R(G-node)).

Adding a new node to a sequential data-structure generally requires reconfiguring the links to represent changes in the logical relationships. Each decision (e.g. IF statement) in the sequential algorithm can be modified to use the range function R() to produce or adjust the proper relationships and position the nodes within the order of the data-structure by range.

All processors may perform this function simultaneously and blindly; however, in the event that two G-nodes with overlapping Ranges collide, the Resolve-Range-conflict() function may be called; Resolve-Range-conflict() is cooperative. In most respects, this function, Insert-G-node(), is identical to the sequential Insert().

9. Remove-G-node(G-node v). All of the statements made about function (8.) Insert-G-node() also apply to this function with respect to the sequential Remove() function. In most respects, this function is identical to the sequential Remove().

10. Resolve-range-conflict (G-node u,v). This function resolves the problem of overlapping G-node Ranges. A difficulty presents itself if a data-structure creates overlapping Ranges because of non-contiguous data placement. Two G-nodes may try to occupy the same or adjacent positions in the data-structure. If two such G-nodes conflict, then the elements in the G-nodes must be divided between them in such a way that the new ranges calculated for an element arrangement do not overlap. This function may determine ranges and force re-distribution of the element values or it may semi-sort the elements across the nodes u and v forcing re-determination of ranges based on the semi-sort.

11. Remove-from-G-node(G-node v, element y). The remove from global node function is called after the appropriate G-node for element y has been located. This function removes the element y from G-node v. This function may arrange the G-node elements in any way desirable for a given data-structure according to

Rules for Fullness or Ordering Scheme, or this function may simply remove element y from the P-node that contains it.

Preferred Embodiment

Generalized Parallel Method

The following is a generalized parallel method which uses the previously defined functions to create the parallel data-structures. The configurations of the data-structures in question are determined by the Insert() and Remove() functions of the sequential data-structures that have been modified slightly to use G-node Ranges to establish ordinable relationships. The slightly modified functions are called from within functions Insert-G-node() and Remove-G-node(). The indications of steps 1, 2 and 3 to the left of the pseudo-code are the steps explained in the Verbal Description section.

Preferred Embodiment

Function Parallel-Insert (element y):

This function is called by any processor wishing to insert the element y into the parallel data-structure. It is assumed that the first G-node of the data-structure has already been created.

```
Parallel-Insert (element y)
G-node u,v
Step 1 →   v = Find-G-node(y)
Step 2 →   if (there is an empty cell in P-node v)
           then
             place y in P-node v
           else
             Add-to-G-node(v,y)
           end if
Step 3 →   if (G-node v is full)
           then
             u = Split-G-node(v)
             Insert-G-node(u)
             Adjust-G-node-ranges(u)
           end if
END FUNCTION
```

Preferred Embodiment

Function Parallel-Remove (element y)

This function finds and removes a specific value y. Some data-structures remove elements by location (example: priority-queue); in such cases, the Find-G-node() function may be adapted to find the proper location, and then the G-node may be sorted or searched for the appropriate value.

```
Parallel-Remove(elementy)
           G-node v;
Step 1 →   v = Find-G-node(y)
Step 2 →   if (G-node v is found)
           then
             Search-G-node(v,y)
             if (y is found in G-node v)
             then
               Remove y from cell and
               send to proper processor
             end-if
           end-if
Step 3 →   if (G-node v is empty)
           then
             Remove-G-node(v)
             Adjust-G-node-Ranges (u)
           end-if
END FUNCTION
```

It is assumed in the preferred instance that most sequential functions for the adaptable sequential data-structures can be encapsulated in the sequential Insert() and Remove() functions, or that they can be adapted and used in the same manner as those functions. The Parallel-Insert() and Parallel-Remove() functions describe the formation and functioning of the parallel data-structures in their important aspects.

Preferred Embodiment Program Adaptation

Find Function

This section contains a small section of C like Pseudo-code intended to illustrate the simplicity of adapting single processor functions to multiple processor functions. The code is not intended to be in perfect C syntax, only to give the basic concepts that could be used for creating the Find function in parallel form. Nor is the code intended to be the only possible embodiment of the Find function (Find-G-node). This example should also help to illustrate the nature of the "slightly adapted" single processor Insert and Remove functions mentioned previously because the majority of the work done for those functions is the location of the proper position in the data-structure for an element value.

The pseudo-code shown is a Find function for a binary search tree; the concepts expressed are given to be useable for other data-structures as well. The pseudo-code shows that the primary difference between single and multiple processor functions is the replacing of the comparisons of element values with comparisons of G-node-Ranges: it is easiest to illustrate this by taking advantage of operator overloading with regard to the <,>, and == operators -- these operators are assumed to work equally well on element values and G-node-Ranges.

Single Processer Definitions and Pseudo-code

```
struct node_st{
        key_type        key;
        node_st         *leftchild;
        node_st         *rightchild;
}
node_st *Find(node_st *node, key_type element)
{
        if (node->key == element)
          return(node);
        if (node->key < element)
          return(Find(node->rightchild,element));
        else
          return(Find(node->leftchild,element));
}
```

Multiple Processer Definitions and Pseudo-code

```
x = Maximum number of elements per P-node;
struct Range{
        key_type lowerbound;
        key_type upperbound;
}
struct Pnode_st{
        node_number     int;
        key_type        key[X];
        Pnode_st        *leftchild;
        Pnode_st        *rightchild;
        Gnode_Range     Range;
}
Pnode_st *PFind(Pnode_st *node, key_type element)
{
        if (node->Gnode_Range == element)
          return(node);
```

-continued

```
        if (node->Gnode_Range < element)
          return(PFind(node->rightchild,element));
        else
          return(PFind(node->leftchild,element));
    }
```

## Preferred Embodiment

Data Model

FIG. **63** depicts a possible data model for an embodiment of the present invention. Each box is a data entity expressing an aspect of the program design for the embodiment. Each data entity is a set of rules, data-structure, record, other set of data and/or associated maintenance functions used to create the parallel maintenance system. No particular modeling technique is implied.

The data model shown is only one possible model, given to express the parallel system components from a data model perspective. The relationships are described below.

- A.1—indicates a set of range adjustment rules may comprise or relate to multiple sets of range addition rules;
- A.2—a set of range adjustment rules may comprise or relate to multiple sets of range removal rules;
- A.3—a set of range adjustment rules may comprise or relate to multiple sets of range breadth adjustment rules;
- D.1—a set of range determination rules may comprise or relate to multiple sets of range adjustment rules;
- G.1—a set of adjustment need rules applies to many G-nodes;
- G.2—a set of range determination rules applies to many G-nodes;
- G.3—a G-node and G-node Range have a one-to-one relationship;
- G.4—a logical relationship may relate many G-nodes, and a G-node may have many logical relationships;
- G.5—a G-node contains many P-nodes;
- G.6—a set of arranging rules applies to many G-nodes;
- P.1—a P-node contains many data value storage entities or elements;
- R.1—a set of range relation rules applies to many ranges;
- R.2—a set of range relation rules applies to many logical relationships;

Notes on B-tree Section

The two rules (Ordering Scheme and Rule for Fullness) used in this section are not the only possible rules for storing a parallel B-tree or other data-structure on multi-component Dynamic Access Storage Devices such as disk drives. Many other Fullness and Ordering rules may be used (defined), but the essential pattern of the present method remains the same. Well known methods exist for storing information on the locations of data storage blocks. These can be used to store information on the fullness of the parallel B-tree-nodes. A bit-map stored in memory would suffice, as would a bit-map stored in high-speed secondary storage (e.g. a faster, more expensive disk drive than the others used). Many other possibilities exist for the storage of the information; the only requirement is that it allows the determination of the fullness of B-tree-nodes and G-nodes without accessing every drive.

It should also be noted that the preferred embodiment and the data-structures and maintenance routines that result from it function by sending the locations of data-structure nodes between processing elements. This may require a method of storing or calculating the location of the memory or disk space allocated for the different portions of data-structure storage: one such method would be keeping an index to a storage space as it is allocated for nodes one each device; another would be the storage or definition of explicit pointers in each P-node to the other P-nodes within a given G-node; for disk drives, many indexing techniques already exist to perform functions very similar to this. Some such techniques are described in "Operating System Concepts" by Silberschantz and Galvin, Addison-Wesley Publishing, 1994 (fourth edition).

The amount of data stored on the B-trees in these examples is small. It is not reflective of the size of B-trees on real systems. In addition, the nodes themselves are relatively small, and the methods of storing data from node-to-node could make use of additional techniques to improve efficiency. Commonly known techniques for single-processor B-trees include the techniques used for B* trees and B+ trees. Also, the use of overflow blocks and techniques derived from B* and B+ trees could be added to the examples given here.

## Other Embodiments

Definition of G-nodes for the B+ Tree

A variation on the B-tree is the B+ tree. The following describes one embodiment of the parallel B+ tree. The file structures book referenced in this application defines a B+ tree as a B-tree for which data-file record pointers are stored only at the leaves of the tree. This indicates that the definition of the B-tree nodes in a B+ tree takes two forms: one form for the non-leaf nodes and one form for the leaf nodes. The same may be done for the definition of G-nodes and their Ranges for parallelized data-structures. I use the parallel B+ tree to illustrate this concept.

Because the elements (tuples) stored in the B+ tree only contain data-file record pointers at the leaf-nodes of the tree, the G-node Ranges in the non-leaf nodes do not require the storage of actual tuples containing record pointers. This means that the only useful information in the non-leaf nodes is the storage of G-node Ranges: the Ranges are used to locate the desired leaf-nodes. B+ tuples are never inserted into non-leaf nodes and therefore the parallel Ranges need not be defined to contain values. Single values may be stored in the non-leaf nodes to represent non-leaf Ranges (the minimum value of the Range may equal the maximum value of the Range).

The leaf nodes of the B+ tree have G-nodes and G-node Ranges defined in the manner described in previous sections. Insertions of new Ranges into the non-leaf nodes occur at the time of B-tree node splits. All non-leaf Range values are based on the values contained in the leaf-nodes of the tree.

## Other Embodiments

Complex Ranges

More complex range calculations than those described in other sections are possible and justifiable. For example, an additional embodiment of an adapted AVL tree may be created by the use of a different set of range relation rules or range determination rules. The AVL tree previously described herein used range relation rules defined in a linear contiguous fashion: $R(A_{o1o}) < R(B_{o1o})$ if and only if $R(A_{o12}) < R(B_{o11})$ (i.e. Max of A less than Min of B); this produced a distribution of the total data set such that the possible storage of a given value x on a processor was only determinable by locating its G-node Range.

Imagine instead a range function such that the highest order digit is ignored. Thus, a range (#50–#70) could contain

values 150,250,350,165,266,360,370, etc. In addition imagine an Ordering Scheme such that processor 1 contains only values whose first digit is 1, processor 2 only values whose first digit is 2, etc. This combination of range function R() and Ordering Scheme create a parallel structure such that a given value is known to be stored on a given processor i or not at all before search begins (e.g. the value 563 will be found on processor 5 or not at all, the value 828 on processor 8 or not at all, etc.) If leading zeros are assumed, then the combination also creates a data structure composed of ten separate structures, each having its own range of possible values (i.e. (000–099),(100–199),(200–299), etc.).

An advantage gained by the grouping of elements into ranges as described above while simultaneously grouping the elements by G-node Ranges is that the elements are sorted by high order digits and sub-sorted by low order digits and simultaneously sorted by low order digits and sub-sorted by high order digits.

Such sorting may even be useful if the P-nodes related by high order digits are grouped and contained on a single graph, rather than the multiple graphs described in other embodiments herein.

Such complex range calculation as described above shows a more advanced grouping of elements by range than other embodiments described herein. The elements contained in the data structure are organized in two fashions: by high order digits and low order digits. This grouping illustrates an element's or a P-node's membership in multiple complex sets. Another instance (a refinement or improvement of the concept of membership in multiple sets) could provide a P-node membership in a plurality of sets, each set organized for access by different aspects of the data stored (e.g. last, first and middle name, etc.).

FIG. 65 shows nine P-nodes, all related by commonality of complex G-node ranges: the nine P-nodes are all part of a complex G-node. The ranges may of course be stored implicitly and partially calculated by processor number; however, on the diagram, they are explicitly listed. Pound signs indicate wildcards; numeric entries separated by dashes indicate ranges; the first two entries may be combined to form an ordinal range and then further refined by adding the last entry: therefore processor 5, having complex G-node Range (#50–#70,2##,##4–##6) may contain numbers between 250 and 270 whose last digit is between 4 and 6. If the nine processors depicted are on a two dimensional mesh of processors, then each linear array may be accessed according to the common key attribute being sought by a user or system process (e.g. any key being sought between 100 and 199 will be found on processor 1, 4 or 7). The rules for insert (i.e. range relation rules) for the data structure in FIG. 65 are assumed to apply to the "#50–#70" portion of the complex range: that is, the links are configured by that portion of the complex range such that if x>y then R(#x) >R(#y). FIG. 65 represents a parallel data-structure considered to have two dimensions at the processor or storage level; the possibility of more dimensions is implied.

The great variety of combinations offers a wide range possible uses according to the needs of a given system or data-structure.

## Other Embodiments

Dependant Ranges

Imagine a decision-tree used, for instance, to play chess. A given function can identify when a piece on the board is threatened by an opposing piece. This increases the priority of moving that piece. A given node in the tree representing this situation on the board will have a wider range of possible moves than nodes dependant on the given node. Each possible movement of the threatened piece is within the range defined by its identity as a move of the piece and its dependency on other nodes. This range may be distributed to multiple processing elements according to range determination, Rules for Fullness and Ordering Schemes defined for the decision tree algorithm's use in a parallel environment. In this instance, the values and ranges for the nodes are created together, rather than input and inserted or removed.

Addendum

G-node Ranges

The calculation of G-node-Ranges is key to the entire process. Generally, it is simply a matter of determining which nodes on a data-structure most closely determine the positions of the other nodes; that is to say which nodes contain values that determine the values that may be contained in a given node. Partially defined G-node Ranges may frequently be found at the extreme points of the data-structure; for instance, the root and leaves of a heap, or the right most and left most node of a binary search tree, 2–3 tree, or B-tree.

Addendum

Ordinal vs. Ordinable Data

Most of the Examples for this application are given for ordinal data types. However, any data-structure having the capacity of the data values to be grouped into suitable G-node Ranges will be adaptable. If the G-node Ranges can be constructed such that the nodes which branch off from the members of the Range can be said to relate to all of those members in a similar fashion, then the parallel or global links between nodes are justified and will be consistent with the data-structure and/or method rules. Such data-structures and/or methods are adaptable by this process.

Addendum

Use of Space/Merging G-nodes

Because the G-node-Remove() function is only used on sufficiently empty G-nodes, it is possible to have large data-structures with large numbers of partially empty P-nodes; however, the present method is capable of adjustment to make efficient use of space. A G-node-Merge() function to merge two sparsely populated G-nodes into one would be one way to resolve this problem; another would be to alter the Rule for Fullness, changing the lower limit on the number of elements in a G-node to half P and remove P-nodes that break the rule, reinserting their elements.

Addendum

Contiguity of Data Distribution

Non-contiguous data distribution like that of a heap makes difficult the efficient search, and therefore efficient placement, of elements into unique G-node Ranges. One solution to this is the Resolve-Range-Conflict() function; however, for those data-structures that can be forced into a contiguous configuration and thereby make efficient searching possible, this function may not be necessary. Non-contiguous methods of defining ranges or distributing values may also be defined and used for the present invention.

Addendum

Data-Structures/Methods

The data-structures and methods listed in this application are only examples of adaptations from sequential to parallel. Many other data-structures and methods not listed can be adapted through this process. No restriction on types of data stored or manner of storage is implied. Many distributed or

parallel data-structures may be created in accordance with the principles of the present invention. The present invention may also be used to create new data-structures without serial counter-parts.

Examples of Application of Preferred Embodiment

The following two examples illustrate the functioning of two adapted parallel data-structures on a working system. Two examples are given to show that the parallel routines and data-structures may function on a variety of different systems. Specifically, one Example is stored in memory on a parallel-processing hypercube network, and the other is stored on disk. Although the data-structures can be used by any program, whether batch or on-line, the examples illustrate the functioning of the data-structures by assuming multiple users accessing the same system simultaneously.

Example of Application 1

FIG. **66** shows a parallel machine with 128 processors connected by a hypercube network. A powerful machine such as this could serve a great number of users simultaneously, but only three are depicted. Each of the terminals (numbered **1**, **14**, and **127**) have allocated their respective processors **1**, **14**, and **127**, and are conducting on-line accesses to a file located on disk in a hashed file with secondary keys stored in an Adapted parallel data-structure: the keys are stored in the memories of the various processors distributed throughout the hypercube on a parallel m-way search tree. Each processor has 16 Mega-bytes of memory. Each processor stores approximately 1/128th of the file's keys in its local memory. If we assume that the search-tree can store each key using 20 bytes of memory (including pointers, indexes, etc.), and we also assume that each processor uses a maximum of approximately 1 Mega-byte of RAM, then approximately 50,000 keys may be stored on each processor: 50,000×128=6,400,000 keys may be stored in parallel memory. The same tree stored in a single processor's memory would require 128 Mega-bytes of memory and probably force the storage of the tree onto disk. In addition, each user on the system may search the tree simultaneously: little or no queuing will result from simultaneous accesses to the tree, unless more than 128 users are logged on.

(Note that FIG. **38** used for this example was designed for 3 processors in the General Example: the key ranges and size of the tree are accordingly small, and the processor numbers illustrated are different.) If we assume that the processors **1**, **14**, and **127** contain the search-tree nodes depicted in FIG. **38**, then User **1** could request key **56**, User **14** could request key **15**, and User **127** could request key **10** simultaneously. Each processor would then access two nodes of its own locally stored tree to reach the bottom level, send requests for keys to other processors as necessary, and receive replies. The same values stored on a single-processor tree would require more accesses (a taller tree) and queuing. In either case the disk could be accessed after retrieval of the keys from the search tree, and the users would receive the appropriate records from disk.

Example of Application 2

FIG. **67** shows three terminals connected to a server, the server is connected to three disk-packs. A parallel B-tree distributed amongst the three disk-packs can be accessed simultaneously by each user. If Users **1**, **2** and **3** all make requests to access the B-tree index at the same time, then the server would have to queue these requests and distribute them among the disk-packs one-at-a-time. However, disk access is much slower than memory access, so the queuing and distribution of the requests in the server's memory might take 10 microseconds per request. Therefore, the last request to disk would be made 30 microseconds after the user made the request. If we assume that each key is stored on the fourth level down in the B-tree, and we assume that each disk access requires 10 milliseconds, then the last request for a key is fulfilled 40 milliseconds +30 microseconds after the request. If the B-tree were stored on a single disk, then in the worst case, each key request would take 10 milliseconds ×4 tree levels ×3 user-requests =120 milliseconds + queuing time.

The precise make-up of the two systems described above differs, but in each case the work is successfully distributed amongst processing-elements, giving better response time. The times suggested and precise make-up of the systems depicted are given only for the purpose of example. The times given are estimates and the calculations are simple illustrations of the functioning of the types of systems that could make use of the Adapted data-structures.

Conclusion, Ramifications and Scope

Thus the reader can see the results of combining the various aspects of this method of creating and using parallel data-structures. The present invention provides a great variety of possible combinations of rules for fullness of nodes, range determination, parallel and global node definition, and data distribution such that each aspect of the invention, in addition to others not listed, may be used in combination with one or more of the others, or alone, to enhance performance of the parallel data structures and define new data-structures, including parallel forms of serial data-structures and many others.

The combinations of components in the embodiments herein are not the only combinations possible. Not only are different combinations possible, but different instances of the components themselves, such differences exemplified by the various rules for fullness, ordering schemes, and range calculations described and contrasted in this application, though not limited to those descriptions or those components.

While my description above contains many specifics, these should not be construed as limitations on the invention, but rather as an exemplification of preferred embodiments thereof. Many other variations are possible. Accordingly, the scope of the invention should not be limited to the embodiments illustrated; the scope of the invention should be determined by the appended claims and their legal equivalents.

I claim:

**1**. A method of maintaining order for data on a computer system by creating a parallel data-structure, said data stored on one or more memory storage means, accessed by one or more processing elements, said order represented either explicitly or implicitly as a graph or graphs containing nodes that represent sets of data values grouped into ranges and incident links that represent logical relationships between said sets of data values, the nodes and links either explicitly or implicitly stored on said memory storage means, said memory storage means and said order maintained by said processing elements,

i. wherein said memory storage means is divided into logically corresponding storage units or partitions, said partitions defined by a parallel storage location of said nodes on said memory storage means, and

ii. wherein one form of said logical relationship is a serial or local relationship relating two Or more differing said

ranges to each other according to their differences, said local relationships relating said ranges within said partition of said memory storage means, and

iii. wherein a second form of said logical relationship is a global relationship relating two or more similar said ranges to each other according to their similarity or commonality, said global relationships relating said ranges between multiple said partitions, and

iv. wherein said nodes form individual nodes having said local relationships with other said individual nodes with different said ranges, said individual nodes also referred to as parallel nodes, and

v. wherein said nodes form global nodes comprising multiple said individual nodes having said global relationships with other said individual nodes, said global nodes thereby comprising multiple said individual nodes with the similar or common said ranges, each said individual node within a given said global node having the common range, such that said global node is a composite of said individual nodes,

the method comprising the steps of:

a. determining said ranges for said sets of data values,

b. assigning said ranges to said nodes and assigning said sets of data values to said nodes by determining the ranges into which they fall,

c. positioning said individual nodes within said order using said links by determining said local relationships and said global relationships between said ranges,

d. storing said nodes in different portions of said memory storage means such that said ranges with said commonality are stored on multiple said individual nodes, each said individual node with the common said range stored in a different said portion thereby defining said partitions and said global nodes comprising a plurality of said individual nodes, and thereby storing said local relationships as explicit or implicit said links within said partition and said global relationships across multiple said partitions,

whereby a combination of the local and global relationships creates a composite global data-structure comprising multiple serial or local data-structures, and whereby said data is maintained in said order on each of said partitions in a uniform manner and on all of said memory storage means combined, and a plurality of system processes are enabled to access the data values simultaneously by accessing said individual nodes in a given said partition of choice, thus gaining access to said global node having desired said range and to the global data-structure as a whole.

**2**. A method as recited in claim **1** wherein said order is expressed as a plurality of separate said graphs, each said graph stored separately within said memory storage means, each said graph arranged by arranging rules of an adapted sequential data-structure, thus creating said parallel data-structure capable of the same functions as said sequential data-structure in a distributed environment.

**3**. A method as recited in claim **2** wherein each said individual node contained in a given said global node has an identical said range to all other said individual nodes in the given global node and wherein all the logical relationships between all said individual nodes belonging to the given global node and all said individual nodes belonging to another said global node are identical.

**4**. A method as recited in claim **2** wherein said memory storage means is composed of a plurality of disks, and said order is defined by a set of rules for maintaining a serial b-tree as adapted to function using said ranges in a parallel environment, thus creating a plurality of b-trees located on said disks, each said b-tree represented as a separate said graph composed of said individual nodes, every said individual node contained on a given said portion of said disks belonging to a different said global node from all other said individual nodes contained on the given portion of said disks.

**5**. A method as recited in claim **2** wherein the plurality of separate said graphs is created and maintained by the present method and each of said separate graphs has an identical structure as every other said separate graph, said structure defined by an identical positioning of each said individual node contained in a given said global node to each other said individual node contained in each adjacent said global node, that is, each said individual node belonging to the given said global node holds the same position within each said separate graph as every other said individual node belonging to the given said global node holds in its said separate graph, whereby each said separate graph is identical in form and function to each other said separate graph and is thus able to function as a separate data-structure on a single said processing element and is also able to be combined with the other said separate graphs and function as said parallel data-structure on multiple said processing elements.

**6**. A method as recited in claim **1** further employing the steps of:

a. identifying where said range is too broad for a given said global node thereby indicating a need to split said range,

b. upon the indication of need to split said range, splitting said range by performing the range determination on the range being split and adjusting adjacent said ranges as necessary thereby creating at least one new said range, assigning the new range or ranges to a new said global node or nodes and performing the positioning to position the new nodes and existing nodes as necessary using said links thereby adding the new nodes to said order and maintaining said order,

c. identifying where said range is too narrow for a given said global node thereby indicating a need to broaden said range,

d. upon the indication of need to broaden said range, performing the range determination to adjust said ranges for adjacent said ranges as necessary, removing the global node containing the too narrow range if necessary, and performing the positioning as necessary to reconfigure said links for remaining said nodes thereby removing appropriate said nodes from said order and maintaining said order,

e. upon the indication that said range or ranges are too broad or too narrow, adjusting said range or ranges by performing the range determination thereby adjusting said range or ranges to proper breadth,

whereby said order is manipulated using said ranges, and said logical relationships are manipulated as necessary to change data storage patterns while maintaining said order of said data.

**7**. A method as recited in claim **6** wherein the range split is performed by creating a new dependent range, said new dependent range based on the range being split, at least a portion of said new dependent range being beyond the range being split thus representing an extension of the range being split and narrowing the range being split by combining the ranges,

whereby the range being split is narrowed by combining the range being split with said new dependant range, the com-

bination of two ranges representing the combination of two restrictions and therefore becoming more restrictive or narrower.

8. A method as recited in claim 6 wherein the identification that the range is too broad is achieved by a rule for fullness of said global nodes that employs a measurement of the number and positions of the data values within said global node, the identification of the too broad range defined by an excess of the data values, said excess indicating that said global node is sufficiently full for the range split and thus for the addition of the new node, and wherein the identification that the range is too narrow is performed by the measurement of the number and positions of the data values within said global node, the identification of the too narrow range defined by an insufficiency of the data values, said insufficiency indicating that said global node is sufficiently empty for the removal of the node, and further employing the steps of:

a. locating a proper said global node with a proper said range to contain desired said data values by traveling along said links and choosing a path through said links, said path determined by using said ranges assigned to said global nodes,

b. upon determination of the proper global node, determining proper said individual node within the proper global node to contain a given said data value,

c. upon determination of the proper individual node, adding or removing said given data value to or from the proper individual node thereby adding or removing the given data value to or from the proper global node,

d. upon the addition or removal of the given data value, determining if the proper global node is sufficiently full or sufficiently empty,

e. upon determination that the proper global node is sufficiently empty for the global node removal, performing the global node removal wherein the global node removal redistributes the data values as necessary within their respective said ranges,

f. upon determination that said global node is sufficiently full for the global node addition, performing the global node addition wherein the global node addition splits the range of the sufficiently full global node and redistributes the data values as necessary within their respective said ranges thereby splitting the sufficiently full global node and adding said new global node or nodes to said order.

9. A method as recited in claim 8 wherein said parallel data-structure is adapted from a set of ordering rules of a sequential data-structure and wherein said parallel data-structure is maintained by said processing elements as controlled by a parallel maintenance process adapted from a sequential algorithm for maintaining said sequential data-structure by utilizing the same said ordering rules applied to said ranges rather than applied to individual said data values.

10. A method as recited in claim 9 wherein the data maintained are ordinal and wherein said ranges defined for said global nodes are unique, non overlapping said ranges covering the expanse of said data.

11. A method as recited in claim 10 wherein the method is used to maintain key values on a distributed database that are accessed by said plurality of system processes or a plurality of users.

12. A method as recited in claim 1 wherein said global relationships are expressed by a specific said parallel storage location of said nodes within said memory storage means such that a first memory address allocated for a first said individual node is used to derive a second memory address allocated for a second said individual node within the same said global node,

whereby locating one said individual node within a given said global node enables said processing elements to easily derive the locations of other said individual nodes within said global node.

13. A method as recited in claim 1 further employing a set of rules for arranging the data values within said global nodes to provide efficient locating means to locate within said memory storage means an exact said individual node contained in said global node that could contain a given said data value within said range.

14. A machine to maintain an order for data on a computer system containing one or more processing means, one or more memory storage means, and communication means linking said processing means and said memory storage means to form said computer system comprising:

a. range determination means to group said data into ranges, each said range capable of being arranged in a sequence or sequences with other said ranges such that said range determination means groups said data into multiple said ranges and such that said sequences between said ranges thereby arrange the data said ranges contain,

b. distribution means to subdivide and distribute each said range to subsets that define subdivisions stored on multiple parallel or individual nodes on said memory storage means,

c. composite global nodes containing the distribution of a given said range, said global node comprising multiple said individual nodes, each said individual node storing a portion of said range defined by said subdivision,

d. relation means to define logical relationships between said global nodes and logical relationships within said global nodes by said ranges, wherein said logical relationship between said global nodes is defined by a difference between said ranges, such that if said relation means compares a given said global node with another said global node or their component said individual nodes, then said relation means achieves equivalent comparison results indicating said difference between said ranges, and wherein said logical relationship within said global node is defined by a commonality, such that component said individual nodes contained within said global node all have the same said logical relationship indicating said commonality in said range, such that said processing means are enabled to arrange said individual nodes with each other using said differences and enabled to arrange said individual nodes within said global nodes using said commonality,

whereby said order is expressed by grouping said data into said ranges and defining said logical relationships between said ranges, and whereby said ranges are able to be distributed within said computer system creating an arrangement of said data providing the order for said data such that it is easily accessed and maintained by multiple system processes.

15. A machine as recited in claim 14 wherein said subdivisions are grouped into separate sets, each said set having its own valid said logical relationships between said subdivisions and therefore between said ranges, said individual nodes, and said global nodes, each said separate set defining a separate graph stored on a division of said memory storage means, thereby creating a plurality of said separate graphs, each said separate graph expressing said order, and all of said separate graphs together expressing

said order thereby creating a parallel data-structure, and further including:

a. individual access means providing an access to an individual said separate graph as an individual expression of said order, said separate graph accessed as a valid separate data-structure, and

b. parallel access means to access multiple said separate graphs together as parallel expressions of said order wherein the access to the individual said separate graph enables access to other said separate graphs, thereby accessing multiple said separate graphs together as said parallel data-structure,

whereby a plurality of said processing means, system processes or users are enabled to efficiently access said data through said separate graphs using separate access paths and accessing separate parts of said memory storage means to achieve consistent results.

**16**. A machine as recited in claim **15** further comprising:

a. range measurement means to determine if said ranges need adjustment providing said range determination means with cause to regroup said data,

b. range addition and removal means to add new said ranges and remove old said ranges to and from said graphs wherein said old ranges are deleted or merged with other said ranges, and said new ranges are derived or split from said old ranges and added in addition to said old ranges, and

wherein said range measurement means determine if the ranges must be added or removed, said range addition and removal means add or remove said ranges, said distribution means redistribute the data defined by said ranges as necessary, and said relation means reconfigure said separate graphs by adjusting the logical relationships between said ranges,

whereby said processing means are enabled to alter a configuration of said parallel data-structure while maintaining said order.

**17**. A machine as recited in claim **15** wherein said distribution means distributes the data to provide said processing means a plurality of dynamically chosen access paths to a given said subdivision or distributed part of range for use by the parallel and individual access means,

whereby said processing means are enabled to choose freely which said separate graph to use for access, accessing a chosen said separate graph until a given said distributed part of range is required, and

whereby said processing means are enabled to efficiently distribute work through the free choice of which said separate graph to use for access to said data.

**18**. A machine as recited in claim **16** wherein said individual access means searches a given said separate graph for a desired said range thereby identifying a proper said global node to contain the desired range whereupon said parallel access means locates a proper said subdivision or subdivisions within said proper global node, said proper subdivisions partially or completely containing the desired range, the identification of the desired range allowing access to desired said data whereupon said processing means uses said data and may therefore have need to alter said order, the alteration of said order is accomplished by using said range measurement means and said range addition and removal means thereby creating a parallel maintenance program executed by said processing means for maintaining said parallel data-structure.

**19**. A machine as recited in claim **16** wherein the range addition means divides an existing said range into sub ranges thereby creating said new ranges, and said distribution means includes an efficient ordering scheme to redistribute said data contained in the existing range to the new range or ranges, thereby creating one or more new said global nodes.

**20**. A machine as recited in claim **18** wherein said parallel maintenance program is adapted from a serial maintenance program for maintaining a serial data-structure, said parallel maintenance program creating said parallel data-structure and utilizing said ranges such that it functions as the adapted serial data-structure in a parallel environment.

**21**. A machine as recited in claim **14** wherein said memory storage means comprises a plurality of memory storage units and wherein said individual nodes are distributed among said plurality of memory storage units and linked by said relation means to form a parallel data-structure.

**22**. A machine as recited in claim **21** wherein said processing means comprises a plurality of processing elements, each said processing element containing a maintenance program for controlling said memory storage means, each said processing element able to control one said memory storage unit at a time and able to cooperate with other said processing elements to control multiple said memory storage units using said maintenance program, the plurality of maintenance programs thereby combining to form a parallel maintenance program,

whereby said parallel maintenance program controls and orders said data through control of said parallel data-structure.

**23**. A machine as recited in claim **22** wherein said parallel maintenance program is adapted from a serial maintenance algorithm, said parallel maintenance program functioning through the use of said ranges, said ranges used as parallel embodiments of the data used in said serial maintenance algorithm.

**24**. An article of manufacture for a computer system, said computer system comprising a memory means and processing means, said processing means comprising one or more processing elements, said processing elements able to access said memory means as one or more logically corresponding storage locations or memory units, said article controlling an ordering of data on said computer system through a parallel storage of said data defining a parallel data structure, said article comprising:

a. range determination rules that enable said computer system to group said data into sets according to ranges of said data, said range determination rules able to define multiple said sets with equivalent said ranges,

b. data storage entity definition rules that enable said computer system to define data storage entities that contain part of said data as defined by said range,

c. parallel node definition rules that enable said computer system to define parallel nodes, said parallel nodes containing one or more said data storage entities, said parallel nodes defined by said ranges indicating the data values that said parallel node is able to contain,

d. composite global node definition rules that define global nodes as composites of said parallel nodes, said global nodes comprising multiple said parallel nodes with a sufficient commonality in said ranges, said parallel nodes having said commonality in said ranges being therefore within the same said global node, said parallel nodes having a difference in said ranges between said parallel nodes being therefore within separate said global nodes where said difference produces sufficient distinction between said sets, said parallel nodes within the same said global node stored on logically corresponding said memory units,

e. range relation rules that enable said computer system to logically relate said ranges and thereby relate said sets, said data and said parallel nodes, said range relation rules determining said commonality and said difference,

whereby said ranges are logically related to each other thereby relating said sets, said parallel nodes, and the data values, and

whereby the relations between said parallel nodes create a plurality of serial data structures linked by the commonality of ranges that defines said global nodes, thus expressing said ordering of data by creating the parallel or global data structure as a composite of said serial data structures, and thus providing parallel and global means to control the data structures.

**25**. An article as recited in claim **24** further including:

a. global node creation means that creates and defines said global node by grouping together said parallel nodes that are related by said commonality in ranges,

b. global node relation rules that utilize said range relation rules to logically relate said global nodes to each other,

whereby said computer system is enabled to globally manipulate said global nodes on said memory means.

**26**. An article as recited in claim **25** further including:

a. adjustment need rules that determine a need for adjustment to said ranges to maintain said order for said data,

b. range adjustment rules that enable said computer system to adjust said ranges, changing the breadth of said ranges,

wherein said range relation rules are used to adjust the logical relationships to appropriately relate the adjusted ranges,

whereby said processing means are enabled to alter a first expression of said order to produce a second expression of said order while maintaining the rules that define said order for both of the expressions, and

whereby changing the data organized in said order results in a change in a given expression of said order while maintaining the rules that define said order.

**27**. An article as recited in claim **26** wherein said range relation rules further define said commonality to produce equivalent comparison results indicating said commonality when comparing one said parallel node within a given said global node to any of said parallel nodes within the same said global node, and further define said differences to produce equivalent comparison results indicating said differences when comparing one said parallel node within a given said global node to any of said parallel nodes within a separate said global node, and wherein said range adjustment rules contain range addition and removal rules to add new said ranges to said order and remove old said ranges from said order, adding new said parallel nodes and removing old said parallel nodes as necessary, and adjusting the logical relationships as necessary.

**28**. An article as recited in claim **26** wherein said computer system defines said order by using said commonality and said difference to create a parallel expression of a serial data structure with its own rules of ordering, thus defining said parallel data structure, said parallel data structure comprising a plurality of separate said serial data structures related to each other by said commonality in ranges and configured by the rules of ordering said serial data structures.

**29**. An article as recited in claim **26** wherein said computer system defines said order by using said commonality and said difference to create a plurality of separate data structures stored separately on said memory units, each said separate data structure identical in configuration to each other said separate data structure.

**30**. An article as recited in claim **27** wherein the range addition is accomplished by splitting said old range into two or more said new ranges, said new ranges being equal to said old range when combined, said new ranges defined such that each has an ordinal range relationship to each other, and wherein said range relation rules relate the ranges in said order to each other by said ordinal range relationship.

**31**. An article as recited in claim **30** further including:

a. find global node means by which said order is searched for a desired said range by using said range relation rules,

b. add to global node means that adds the data values to said global nodes,

c. remove from global node means that removes the data values from said global nodes,

wherein locating the desired range allows access to a proper said global node to contain a given value of said data, and upon the locating, the data values are added to or removed from the proper global node altering the global node contents as necessary, and said adjustment need rules determine if the alteration of the global node contents results in said need for adjustment, whereupon said ranges are adjusted and the relationships are altered as necessary.

**32**. An article as recited in claim **31** wherein the logical relationships, said find global node means, the addition of new parallel nodes and the removal of old parallel nodes are adapted parallel versions of a search algorithm, logical relation rules, node or data addition rules and node or data removal rules of a serial data structure,

whereby said parallel data structure is a parallel version of said serial data structure created and maintained in a parallel or distributed environment, said parallel data structure accomplishing the same goals as said serial data structure.

**33**. An article as recited in claim **31** wherein said ranges are non overlapping ranges that relate to each other in the same fashion as the data values properly contained in said ranges such that said range relation rules express the similarity in relationships to create said parallel data structure.

**34**. An article as recited in claim **24** wherein said range determination rules and said range relation rules create the range definitions with a wide variety of uses, the range definitions creating complex ranges, said complex ranges defining complex sets, said complex ranges calculated using said data in such a way that a given value of said data can have membership in multiple said complex sets, said multiple complex sets intersecting each other where said multiple complex sets contain the data value with membership in said multiple complex sets.

**35**. An article as recited in claim **34** wherein said complex ranges are used to create said parallel data structure such that it has at least two dimensions, and wherein each said complex set is organized for access by a different aspect of the data values stored in said parallel data structure.

**36**. An article as recited in claim **35** wherein said parallel nodes related by said commonality in ranges are said complex sets of said parallel nodes, and wherein each said complex set of parallel nodes creates a complex said global node, said complex global nodes related to each other by said range relation rules.

* * * * *